ACC Programmer's Reference Guide

Edition 1

HP 9000 Networking

Manufacturing Part Number : Z7345-90008

E0204

© Copyright 2004 Hewlett-Packard Company, All rights reserved.

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY 3000 Hanover Street Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Copyright Notices. ©copyright 1983-2000 Hewlett-Packard Company, all rights reserved.

Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.

©copyright 1979, 1980, 1983, 1985-93 Regents of the University of California.

This software is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

©copyright 1980, 1984, 1986 Novell, Inc.
©copyright 1986-1992 Sun Microsystems, Inc.
©copyright 1985-86, 1988 Massachusetts Institute of Technology.
©copyright 1989-93 The Open Software Foundation, Inc.
©copyright 1986 Digital Equipment Corporation.
©copyright 1990 Motorola, Inc.
©copyright 1990, 1991, 1992 Cornell University
©copyright 1989-1991 The University of Maryland
©copyright 1988 Carnegie Mellon University
©copyright 1989-1997 Data Connection Limited

Trademark Notices UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X Window System is a trademark of the Massachusetts Institute of Technology.

MS-DOS and Microsoft are U.S. registered trademarks of Microsoft Corporation.

OSF/Motif is a trademark of the Open Software Foundation, Inc. in the U.S. and other countries.

Printing History

The manual publishing date and part number indicate its current edition. The publishing date will change when a new edition is published. Minor changes may be made without changing the publishing date. The manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

First Edition February 2004 Release 1.0

Related Documentation

The documentation available for the Multiprotocol ACC family of products includes the following hardware and software manuals:

Hardware Manuals

- *8 Channel PCI ACC Multiplexer Hardware Installation and Reference Manual*

Software Manuals

- *ACC Installation and Configuration Guide*
- *ACC Utilities Reference Guide*
- *ACC Programmer's Reference Guide*
- *ACC Error Guide*
- *HDLC Frame Protocol User's Guide*
- *ACC X.25 Protocol User's Guide*
- *ACC X.25/ISDN Data Analyzer User's Guide*
- *ACC HDLC/LAP-B (ABM) Protocol User's Guide*
- *HDLC-NRM (SDLC) Protocol User's Guide*
- *X.25/ACC Installation and Configuration Guide*
- *X.25/ACC Update Guide*
- *X.25/9000 User's Guide*
- *X.25/9000 Programmer's Guide*

Contents

1. ZCOM Subsystem

Introduction 12
ZCOM Software Overview 13
ZCOM Concepts 17
Program ZLUs 19
Terminal ZLUs 19
User Interface 21
References 22

2. ZCOM Message Handling

Overview 24
ZLU Definition 25
Program ZLUs 25
Terminal ZLUs 25
Mapped ZLUs 26
Message Queuing 27
Priorities 29
Multiplexing 30
Outbound Multiplexing 31
Inbound Multiplexing 32
Terminal State 34
Error Handling 35

3. ZCOM Tables and Data Structures

Introduction 38
Memory Organization 39
Differences in 32-bit and 64-bit Data Structures 40
ZCOM Header Structure 42
Node Entries Table 55
ZLU Tables 59
Individual ZLU Entries 59
Logical Terminal Tables 62
Physical Terminal Tables 68
Interface Table 77
Response Records 98

Contents

Queue Header 101
Data Buffer Pool 106
Organization of Buffer Pool 106
Message Header 112
Message Header (zmhd_type) 112
Message ID Header (zmid_type) 113
Message Request Header (zmrq_type) 116
System Event Message (zevent_type) 119
ZCOM Kernel Data 124
NCARD 124
HPA Table 124
SYS 128
HP 132
LINFO 133

4. ZCOM C I/F Library Routines

Introduction 136
Initialization call 137
Interface configuration calls 138
ZLU configuration calls 138
Data storage allocation calls 138
ZLU data transfer calls 138
ZLU information calls 139
Utility routines 139
ZLU Definition 140
Man Pages 141
GETDEVICE (3X) 142
LTFIND (3X) 147
LTQDGET (3X) 151
LTQDPUT (3X) 155
MAKEZLUNAME (3X) 159
PTFIND (3X) 162
ZCLOS (3X) 168
ZCNTL (3X) 170
ZCOMERROR (3X) 178
ZCOMLNAME (3X) 181

Contents

ZCOMLOG (3X) 183
ZCOMSTATUS (3X) 188
ZCONFIG (3X) 190
ZEVENT_RCVR (3X) 226
ZGET_SHRCVR_LIST (3X) 233
ZINFO (3X) 236
ZINIT (3X) 240
ZLTMG (3X) 243
ZLTMX (3X) 245
ZLTQUEUE (3X) 248
ZLTSTORE (3X) 252
ZLTUP (3X) 257
ZLUOPEN (3X) 260
ZMAPR (3X) 264
ZNAME (3X) 266
ZOPEN (3X) 270
ZPEEK (3X) 274
ZPORT (3X) 276
ZPTUP (3X) 287
ZQMVE (3X) 289
ZQSZE (3X) 292
ZREAD (3X) 296
ZRNTIMER (3X) 304
ZSEND (3X) 307
ZSETQL (3X) 316
ZSET_RCVR (3X) 321
ZTIMR (3X) 326

Contents

1 ZCOM Subsystem

Introduction

The ZCOM Subsystem consists of software and firmware that provide the base set of features for the ACC Product family. It consists of:

- HP-UX interface drivers that communicate with the ACC Mux cards
- the ZCOM Mux Kernel which is an operating system that is downloaded to the ACC Mux cards
- a set of daemon and utility programs that provide and facilitate access to the ZCOM Subsystem
- a library of routines that provides a comprehensive user interface to the ZCOM Subsystem.

The ZCOM Subsystem also provides a powerful framework for experienced data communication programmers to develop customized protocols for specialized data communication applications.

This document describes the programmatic interface to the ZCOM Subsystem. It describes the ZCOM Subsystem tables, buffers, and routines, and how they are used to achieve program-to-device and program-to-program communication. It also contains information required when writing application programs to take advantage of ACC product features, such as:

- remote node access

  This feature allows programmatic requests to be routed to either a Mux card that is connected to the system on which the request is made, or to a program or Mux card on a remote system.
- dual interface Mux panel configurations

  This feature allows two systems to connect to a single Mux panel.
- multiple protocol support
- true multi-thread programming support
- 64-bit programming support

ZCOM Software Overview

Figure 1-1 on page 14 presents an architectural overview of the ZCOM Subsystem.

The ZCOM application interface provides low level access to the ZCOM Subsystem. Application programs make requests via the ZCOM library routines which are then communicated to the firmware on the ACC Mux card via the HP-UX drivers.

The firmware which runs on the ACC Mux card includes the ZCOM Mux kernel software plus ZCOM protocol modules. The ZCOM protocol modules provide the protocol specific firmware that runs on the ACC Mux card.

Note that the highlighted area in Figure 1-1 shows which software components are provided by the base ZCOM Subsystem. Only protocol modules used for loopback testing, and for monitoring the health of the ACC Mux card and panel are provided with the base ZCOM Subsystem. Protocol specific modules, such as X.25, are added to the ZCOM

Subsystem as separate products. The Protocol User's Guides provided with these products must be used in conjunction with this Programmer's Reference Guide when writing ZCOM application programs.

Figure 1-1 Overview of ZCOM Software Structure

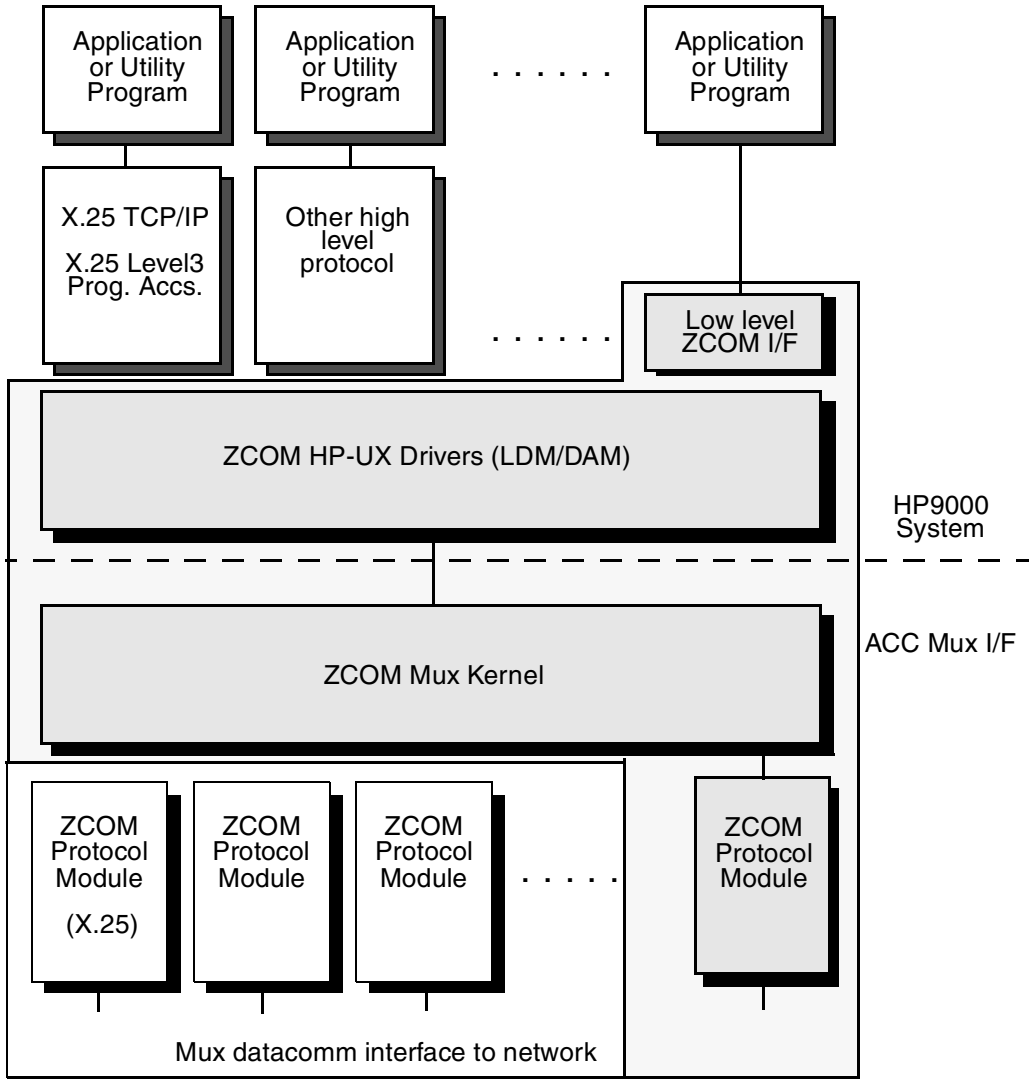
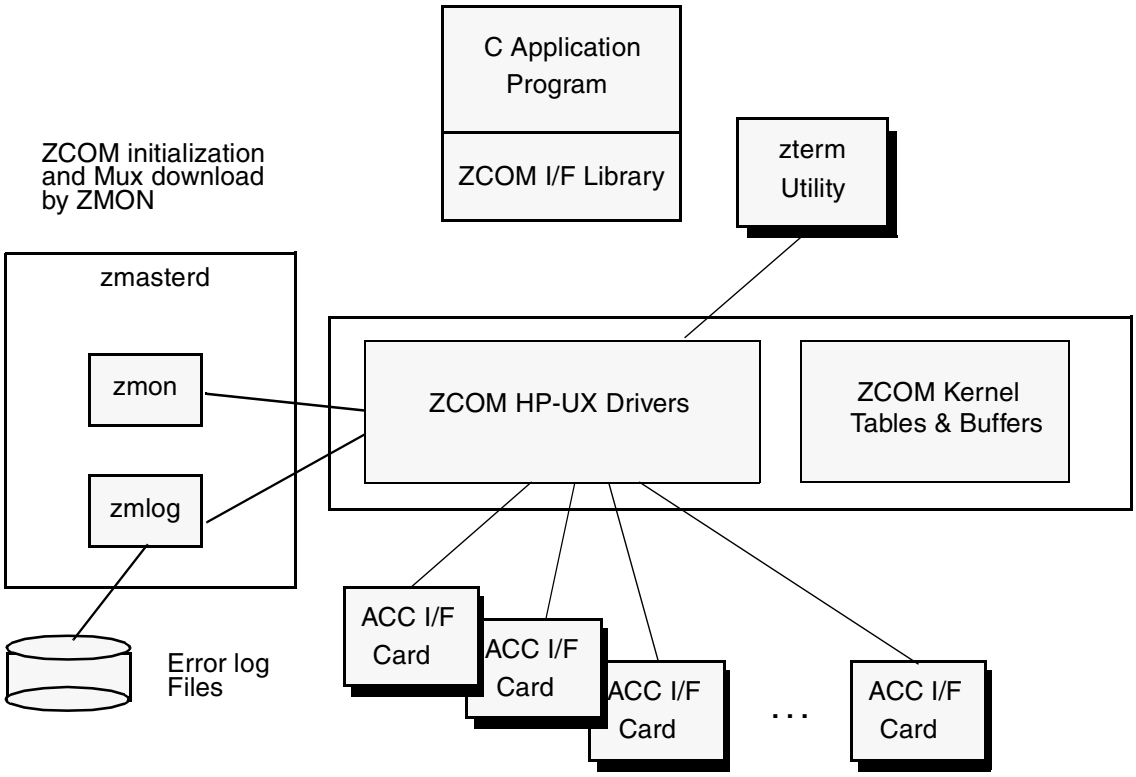


Figure 1-2 presents another view of the ZCOM Subsystem which includes the relationships between the various components of the ZCOM Subsystem:

Figure 1-2 ZCOM Subsystem Components



The ZCOM Subsystem is initialized by the *zmon* daemon which reads a memory image file and downloads appropriate firmware to the ACC Mux interface cards. The memory image file is created by running the *ttgen* utility on an ASCII configuration file which contains information such as what protocol will be used on the various ports of the ACC Mux cards, and protocol specific configuration.

The *zmlog* program is used to log error messages generated by the programs and daemons of the ZCOM Subsystem. The *zmasterd* daemon is used to facilitate starting up and shutting down the daemons in the ZCOM Subsystem, and to automatically recover from daemon failures.

Application programs make calls to the routines in the ZCOM application interface. The same requests that can be made programmatically can also be made interactively by using the *zterm* program.

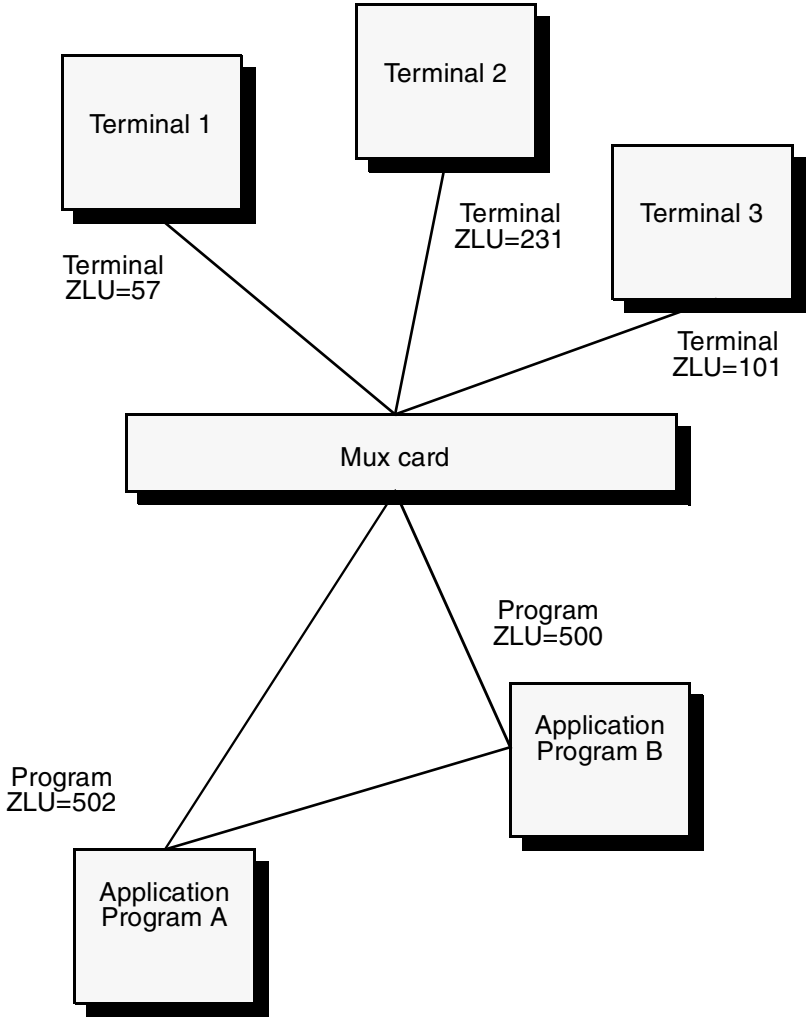
ZCOM Concepts

Throughout this manual, references are made to *terminal ZLU* or just *terminal*. This does not mean a physical terminal device type but is a generic term that can refer to ***any*** kind of physical or logical entity. Examples of physical entities are printers, an HDLC link, terminals, etc. Examples of logical entities might be an X.25 Permanent or Switched Virtual Circuit.

In the ZCOM Subsystem, messages are sent directly from sender to receiver (where the receiver is another program or a terminal ZLU). They are routed using a pointer called a 'ZLU' which stands for 'ZCOM Logical Unit'. ZCOM uses the ZLU for its addressing mechanism. A ZLU may represent a terminal (for instance, an X.25 virtual circuit) or an application program. Messages are sent to and received by ZLUs.

Figure 1-3 on page 18 shows a sample set of program and terminal ZLUs, and their interaction with each other and a Mux card:

Figure 1-3 Definition and Features of ZLUs



To communicate to another program or to a terminal, a program uses ZCOM library calls to write to a program or terminal ZLU. Each program ZLU points to a queue that contains all messages, including timeout and status messages, that were delivered to the program ZLU. These pointers

are maintained automatically by the ZCOM Subsystem. When a message is sent to a ZLU, the data is moved to a system buffer first and the address of the message is moved to the attached input queue.

Program ZLUs

Each program may open one or more program ZLUs which provides an input message queue to that program. Think of a program's input message queue as a mailbox where messages can be delivered. All messages delivered are queued at the end of the input queue for a program in the order in which they were received. Note that the message is not actually delivered to the program, but simply placed into the input queue. The program must issue a ZCOM library call to retrieve the messages one-by-one from the input queue. The method used to cause a message to be placed into a program's input queue will be different depending on whether the message was created by another program or a physical/logical device.

If a message is sent (generated) by another program, then the sending program must specify the ZLU of the program's input queue to place the message into. The receiving program need not do anything for the message to be successfully delivered to their mail box (program input queue).

However, when a message is generated by a terminal ZLU, any program that wishes to receive messages from that device (terminal ZLU) must inform the ZCOM Subsystem first. The program should issue a ZCOM call (zset_rcvr) to tell the ZCOM Subsystem that all data coming from the specified terminal ZLU should be delivered to its input queue. Whenever data arrives from the device, it will be placed into the input queues of all programs that have requested this data.

Terminal ZLUs

Terminal ZLUs represent a physical or logical device and do not have input queues of their own. Any data coming from a terminal ZLU will be delivered to the input queues of all applications that have requested it. If there is no application interested in receiving the data, then the data is thrown away and a message is logged to that effect.

Terminal ZLUs do however have three different output (transmit) queues. When a program sends data to a terminal ZLU, it can select one of three queues to place that data on. Each queue represents a different

level of priority. The *Express Queue* is used for extremely high-priority protocol dependent requests. The *High-Priority Queue* is used for high priority data and the *Low-Priority Queue* is used for low priority data. When a message is generated by an application, the buffers involved are queued to the selected transmit queue and the ZCOM driver is notified. The driver is then responsible for handling the message. Messages on the express queue are sent before those on the high priority queue which are sent before those on the low priority queue.

Because the ZCOM Subsystem itself takes responsibility for the delivery of messages, the sending program is able to specify a number of options when it passes the message to the system. These options include:

- Continue execution whether the message is sent successfully or not.
- Continue execution but report any error status (status messages are queued to the ZLU of the calling program).
- Continue execution; a definite response will be queued to the program ZLU when the send has completed.
- Suspend execution until the send has completed (i.e., acknowledgment that the message was transmitted).

On completion, the message buffer may also be passed back to the sending application program if required.

Each message on the ZLU has a 'type' attribute so that the program reading the ZLU is able to differentiate between status messages, timeout messages and data messages from terminals or other programs.

User Interface

The interface between the application programs and the ZCOM Subsystem is through the ZCOM 'C' Application Programming Interface (API) library. The functions provided by this library are documented in Chapter 4, "ZCOM C I/F Library Routines." All functions within the library are thread-safe and can be used in multi-threaded programs. Both 32-bit and 64-bit applications are supported.

The interactive program ZTERM gives the user access to all the ZCOM routines and can be used to test programs or terminals by sending and receiving messages between them. The program ZSCAN allows the programmer to display the contents of the ZCOM memory areas.

References

The following manuals should be referred to when using the ZCOM application interface:

- *ACC Installation and Configuration Guide*
- *ACC Utilities Reference Guide*

The manuals corresponding to the protocol being used should also be referred to:

- *ACC X.25 Protocol User's Guide*
- *ACC HDLC/LAP-B Protocol User's Guide*
- *ACC HDLC/NRM Protocol User's Guide*

Documentation recommended for protocol developers includes:

- *Z16C35 CMOS ISCC Technical Manual*
  (Zilog P/N DC-8286-01)
- *Z80180/Z180 MPU Microprocessor Unit Tech Manual*
  (Zilog P/N DC-826-03)
- *Z80 CPU Technical Manual*
  (Zilog P/N DC-0029-04)
- *SCC Technical Manual*
  (Zilog P/N DC-8293-01)

2 ZCOM Message Handling

Overview

ZCOM message handling is achieved through logical constructs called ZLUs which maintain queues for messages to terminals and programs. The terminal ZLUs are either defined at ZCOM configuration time using *ttgen*, or can be created programmatically using the Dynamic System Configuration functions; the program ZLUs are allocated dynamically from a pool of ZLUs when requested.

ZLU Definition

There are three types of ZLUs. These are:

- Program
- Terminal (or remote device via a communication link)
- Mapped

Program ZLUs

A Program ZLU may be either a primary ZLU or an auxiliary ZLU. All programs using ZCOM must have a primary ZLU defined. Other ZLUs may then be defined as auxiliary if required. If a message is sent to another program, then the destination ZLU is the other program's primary or auxiliary ZLU.

Terminal ZLUs

Terminal ZLUs are used to direct messages to terminals. Before a program can receive messages from a terminal, a program must set itself up as receiver for that particular terminal's ZLU (using the *zset_rcvr* routine call). Any number of programs can be set up as a receiver for a given terminal at any one time. It is possible, of course, to set up a program to route messages to other programs from the terminal ZLU.

NOTE

Terminal ZLU is a generic term. The actual physical entity associated with a terminal ZLU is dependent on the particular protocol being used. For example, with a TTY protocol a Terminal ZLU would be associated with a physical terminal or terminal like device. However, with a protocol such as X.25, a terminal ZLU might represent a virtual circuit or the HDLC line (X.25 level 2).

Mapped ZLUs

All messages addressed to a particular ZLU will be redirected to the alternate ZLU specified in the *zmapr* routine call if the original ZLU has been mapped to a different ZLU. One use of mapped ZLUs is for providing high availability systems. For example, if a hardware failure was detected in one of the ACC Mux cards, the application could map all of the ZLUs associated with the failed card to a spare unused card in the system.

Message Queuing

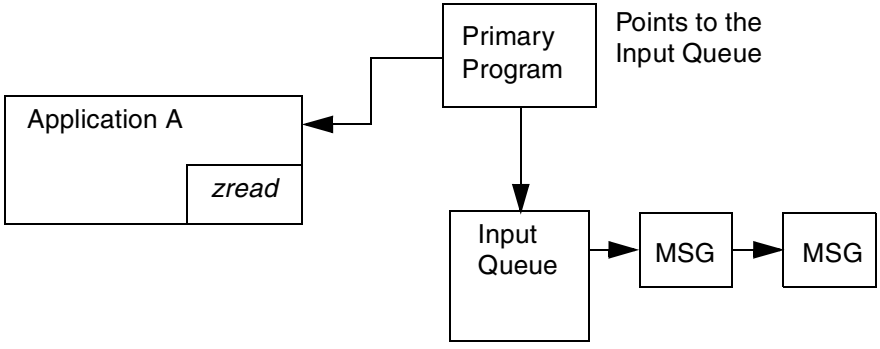
Each program ZLU (either primary or auxiliary) has a single input queue on which messages are received and read on a first-in-first-out basis. Because the ZLU is the input queuing mechanism for timeouts and statuses, as well as for data messages, all programs should request (through the *zopen* routine) at least one program ZLU before using the other ZCOM calls.

The *ztimr* routine may be used to enable or disable the timer on a program ZLU. When enabled, a timeout message is added to the input queue of the ZLU every *n* seconds (*n* being specified as a parameter to the *ztimr* call). If there are already 10 or more messages on the queue, no timer message is queued. Timer processing stops when ZTIMER and ZCLOCK are reset to zero via the *ztimr* routine (a zero "time" value).

The application program uses the *zread* routine to read messages from the input queue. All response and status messages are written to the primary program ZLU. Data messages can be written to either the primary or auxiliary program ZLUs.

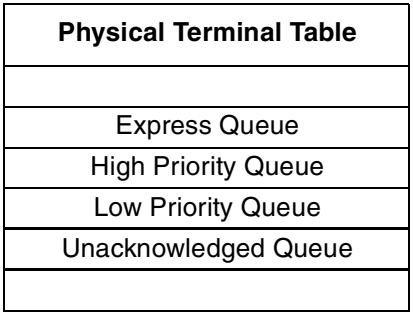
When a *zsend* routine call is executed, the message to be sent is moved to the ZCOM kernel buffer which is queued to the ZLU to which the message was sent. The message is added to the end of the queue for the specified ZLU.

Figure 2-1 Program Message Queueing



Queues for terminal ZLUs are located in the physical terminal tables in the ZCOM memory area as shown in Figure 2-2. The physical terminal tables (one per terminal) maintain four queues for each terminal.

Figure 2-2 Physical Terminal Table



The driver reads messages first from the express queue until there are no remaining express queue messages, then from the high priority queue until there are no high priority messages, at which stage the low priority queue is read. Any further additions to the higher priority queues will be read before the driver continues with the lower priority queues. When the interface card is ready to accept messages, the driver moves the message to the unacknowledged queue and transfers the message to the interface card (through the unacknowledged queue). Both the interface card and the driver use DMA (Direct Memory Access) hardware to transfer the messages. If the number of buffers on the unacknowledged queue is less than the limit specified to TTGEN, more messages will be transferred, otherwise the transfer will be suspended until some outstanding messages are acknowledged.

Once the interface card has transmitted the message and has responded with a definite status, the driver passes the status (if requested to do so) back to the calling program. If the calling program also requires the message buffer to be sent back with the status response, it is picked up from the unacknowledged queue. Once the driver has passed the message to the program, the message is removed from the unacknowledged queue.

Messages in the unacknowledged queues for each terminal are also used in a Mux restart: the messages are moved back to the high priority queue to be resent. The size of the unacknowledged queue is configurable in the TTGEN configuration file.

Priorities

Priorities only apply to messages that are destined for terminals. Messages to program ZLUs are simply added to the end of the input queue for that ZLU.

Messages destined for terminal ZLUs may have either express, high or low priority. The priority for a message is determined by bits ZCOM_ZSEND_LPR and ZCOM_ZSEND_XPS of the *mode* parameter, in the *zsend* and *zcntl* routine calls. If the ZCOM_ZSEND_XPS bit is set, then the message is queued on the express queue in the physical terminal table for that terminal ZLU. If the ZCOM_ZSEND_LPR bit is set, then the message is queued on the low priority queue in the physical terminal table for that terminal ZLU. If neither bit is set, then the message is queued on the high priority queue. This enables background tasks to be sent at lower priorities, without impacting any on-line throughput.

The driver reads the information from the terminal tables, looking at the high priority queue for each terminal and sending the message at the head of that queue. The driver only checks the low priority queue if there are no messages on the high priority queue. This means that any message on the high priority queue will be sent first.

NOTE

The Express Queue should be reserved for Protocol Control events, such as, interrupts or resets. Normally, data should not be sent using the Express Queue.

Multiplexing

The multiplexing feature of ZCOM allows more than one logical terminal to be mapped to a single physical terminal. This mechanism is implemented in ZCOM by allowing messages to and from a terminal ZLU to be intercepted by a program (which is able to alter the data content, e.g., adding or removing headers) and then forwarded on to the destination physical terminal or its receiver program.

Some fields in the logical terminal table and in the physical terminal table are mentioned in the following descriptions; more information on these fields can be found in the logical terminal table and the physical terminal table information in the section on Tables and Data Structures.

The outbound and the inbound multiplexing are separate, so although in most situations the multiplexing program will perform both functions, it is possible to perform only inbound multiplexing or only outbound multiplexing in any program.

The *ltflag* field of the logical terminal table (zltt_type) contains two flag bits ZCOM_LTFLAG_OMX and ZCOM_LTFLAG_IMX, which indicate whether messages to or from the logical terminal respectively, are to be handled by a multiplexing program. These bits are set by TTGEN according to the device type.

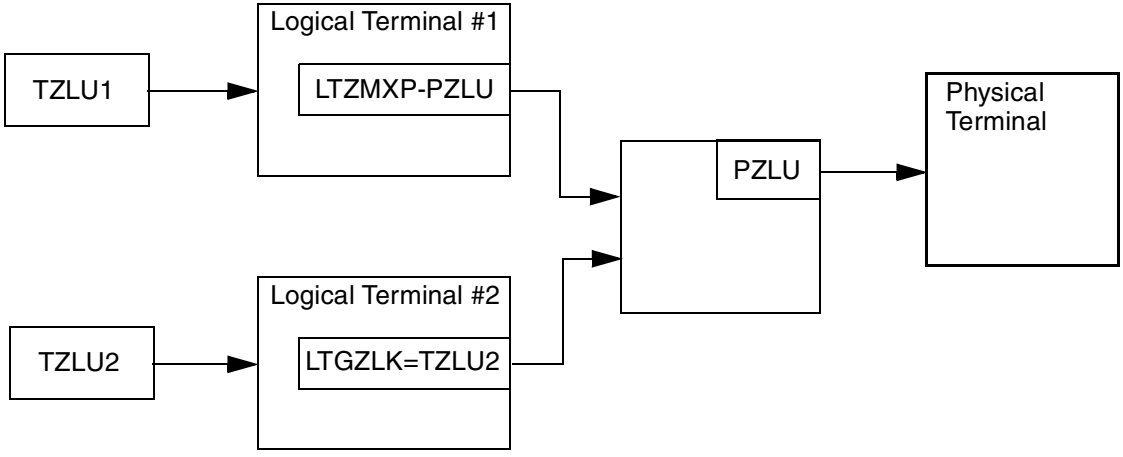
When more than one logical terminal is mapped to a single physical terminal, the *ltgzlk* field (<zltt_type>.ltgzlk) linkage forms a circular list of terminals in the same multiplexed group (i.e., mapped to the same physical terminal). The *ltqzlk* field contains the ZLU number (16-bit) of the next terminal in the group. For a non-multiplexing terminal, this field contains the ZLU number of the current terminal (i.e., pointing back to itself, a special case for circular list).

Outbound Multiplexing

The multiplexing program uses the *zread* call to determine the origin of the message, after which, the program would normally attach a protocol header and forward the message to the destination physical terminal. MZAUXL in the message header (see the chapter on Tables and Data Structures) can be used to save the originating source node and ZLU on the outbound message. The MZSRCE field in the message header is then overridden with the multiplexing program's primary ZLU, hence status and response messages may be routed to the multiplexing program instead of the sender. Routine *zsend* writes to the physical terminal (with bit ZCOM_ZSEND_NOMX set in the *mode* parameter). Routine *zsend* with user header (*mode* bit ZCOM_ZSEND_MHD) must also be used.

The responses to the outbound messages are received by the multiplexing program as type 6 (ZCOM_MSTYPE_RSLT) messages (request code ZCOM_MRQCODE_WRITE [2]). The multiplexing program has to strip the protocol header which it had previously added and forward the response to the ZLU number saved in MSPARE (using *zsend* with bits ZCOM_ZSEND_MHD and ZCOM_ZSEND_NOMX both set).

Figure 2-3 Outbound Multiplexing



If the ZCOM_LTFLAG_OMX bit is set in the LTFLAG word of the logical terminal table, then a logical terminal is enabled for outbound multiplexing. This means that any message sent (by *zsend*) to this terminal ZLU will be formatted as normal (type 4, ZCOM_MSTYPE_LPLT, message with request descriptor block), but instead of being queued to the physical terminal transmit queue, it will be queued to the program ZLU defined by LTZMXP. These fields must have been defined to the system for each terminal by a prior *zset_rcvr* call (*mode* 2, ZcOUTB_MLTPLX) by the outbound multiplexing program during its initialization.

The *zcntl* call works in a similar way to *zsend*. For example, an enable of a multiplexed logical terminal ZLU will queue an enable request (type 4, ZCOM_MSTYPE_LPLT, request code 7, ZCOM_MRQCODE_ENB) to the outbound multiplexing program's input queue. These messages would normally be used to maintain the flags for the logical terminal in LTSTAT. They may also generate a control command to the physical terminal from the multiplexing program.

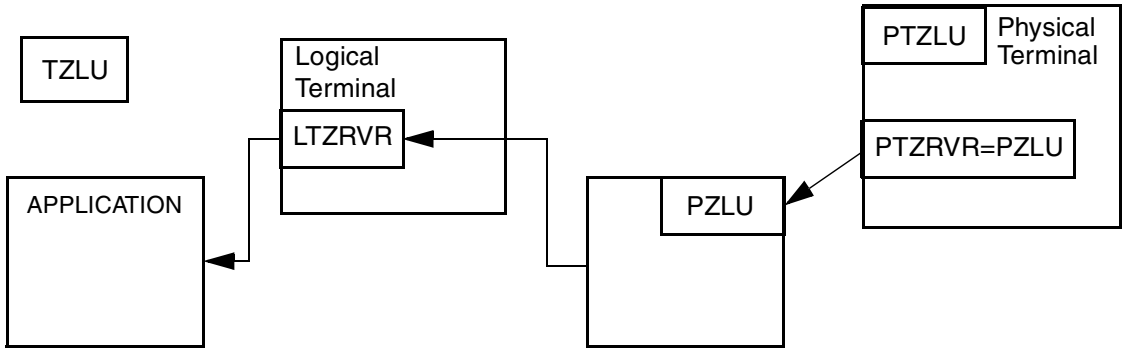
Inbound Multiplexing

If the ZCOM_LTFLAG_IMX bit is set in the LTFLAG word of the logical terminal table, then the logical terminal is enabled for inbound multiplexing. This means that when an application program selects the terminal for input (using **zset_rcvr** default mode 0), **zset_rcvr** will insert the application program's ZLU into LTZRVR (rather than PTZRVR for a terminal without inbound multiplexing).

It is up to the inbound multiplexing program to define itself to the ZCOM subsystem as the receiver for messages from the physical terminal using the *zset_rcvr* (mode 1, ZcINB_MLTPLX) call. This sets PTZRVR in the physical terminal table.

The inbound multiplexing program has the task of receiving all messages from the physical terminal. It determines the origin of the logical terminal from which the messages were sent. Then it processes any of the control messages and forwards the data portion of any data messages to the application receiver.

Figure 2-4 Inbound Multiplexing



Thus, to process an inbound message with multiplexing:

- *zread* (the message type will be type 1, ZCOM_MSTYPE_MSLT - a message from a terminal ZLU).
- The source ZLU is obtained from PTZLU. PTZLU gives the ZLU of one of the ZLUs in a multiplexed group.
- Decode the protocol header to extract the logical terminal address. Use the *ltfind* routine to scan the multiplexed group to find a match of the logical terminal address with LTADDR.
- Build a ZCOM message header where the source ZLU is the ZLU found by *ltfind*, and the destination ZLU is taken from LTZRVR of the logical terminal able.
- Use *zsend* (with bit ZCOM_ZSEND_MHD set) to send the message and the ZCOM header to LTZRVR.

Terminal State

Within the ZCOM subsystem, the terminal state may be either enabled or disabled, activated or deactivated. For normal I/O operations, a terminal ZLU must be both enabled and activated. If a terminal is disabled (regardless of whether it is activated), then no messages can be sent to it or received from it.

If a terminal is enabled but deactivated, then messages can be sent to the terminal, but none can be received. The following table illustrates the different combinations.

Terminal State	SEND	RECV
Enabled/Activated	Yes	Yes
Enabled/Deactivated	Yes	No
Disabled/Activated	No	No
Disabled/Deactivated	No	No

NOTE A terminal state of disabled overrides the activated/deactivated status.

For some protocols (such as the NCR 301 protocol), the protocol handler automatically deactivates the terminal when a valid message is received and until it has been processed. Each time a message is received, the terminal must be reactivated by the receiving program, using the *zcntl* command with the 'activate' request code (request code 9, ZCOM_MRQCODE_ACT).

Error Handling

In the normal functioning of ZCOM, transmission errors on messages to terminal ZLUs are handled by the Mux interface card firmware. The firmware always sends a status back to the driver after it has processed a request - retries are handled internally. The driver checks the mode of the send (i. e., send no wait, report errors, etc.) to determine what to pass back to the requesting program.

If a power failure occurs or the Mux interface card crashes, the driver receives an interrupt from the operating system. This interrupt causes the driver to send a request to ZMON to download the firmware to the particular interface card and to perform a full restart sequence (which also involves re-sending any unacknowledged messages). If the directory */var/opt/acc/dmp* exists, ZCOM will upload the contents of the Mux card memory and store it in a file in this directory whenever a Mux card failure is detected. This provides the ability to analyze firmware failures which can be particularly valuable during the development of custom protocols.

For a detailed description of ZCOM C interface routine error codes, please refer to the *ACC Error Guide*.

3 ZCOM Tables and Data Structures

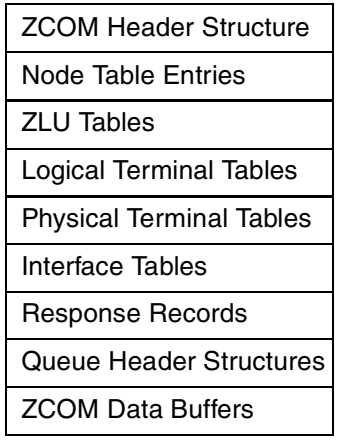
Introduction

This chapter gives a detailed description of the ZCOM subsystem memory organization and table layout. Each table structures is described in overview, followed by a detailed explanation of all of the elements within the table.

Memory Organization

The ZCOM Logical Device Manager (LDM driver) reserves a contiguous block of kernel memory for tables for its use. The maximum size of this memory block is tunable at kernel build time through the ***zcom_mem_size*** parameter in the "system" file used to build your kernel. The default value for this parameter is 4 Mbytes. This memory is allocated relative to a starting pointer. The memory is organized by function into the following groups:

Figure 3-1 Memory Organization



The TTGEN configuration file describes the sizes and initial content of these tables. When TTGEN is run, it produces a memory image file that is the same size and in the same format as this memory layout. The memory image file is loaded into the kernel memory by ZMON via the LDM when the system is initialized. Most of the tables are preset up in the file by TTGEN; however, the response headers, queue headers and data buffers are created and initialized by the LDM.

Note that TTGEN reports the size of the memory block in the "System Information" section of the listing (if a listing is requested). If the tunable parameter ***zcom_mem_size*** is different from the reported size of the memory block, the 'ZCOM Data Buffers' area will be increased or reduced to fit the memory block in the kernel memory during ZCOM subsystem startup.

Differences in 32-bit and 64-bit Data Structures

HP-UX supports 64-bit mode using the *LP64* model. This means the *long* and pointer fields are 64-bits in length. The ZCOM memory data structures are designed to be 32/64-bit neutral by using conditional padding fields. For example, a pointer field is defined as a 64-bit pointer in 64-bit mode. But when the same field is used in the 32-bit kernel, it is a 32-bit pointer **prefixed with a 32-bit padding field** (the padding is inserted in front of the field). The data structures hence are the same sizes in both 32 and 64-bit modes. This gives better control over data alignment in the data structures, and allows 32-bit utilities to manipulate both the 32and 64-bit data structures easily.

But this still does not give full transparency to 32-bit programs that access ZCOM kernel tables on a 64-bit system. When a 32-bit program manipulates a 64-bit *long* or pointer field in a data structure, the program should consider whether the data structure comes from a 32-bit or 64-bit kernel, even though it has the same size and layout. The program should treat the padding fields as hidden fields (same as the data structure alignment gaps inserted by the compiler) and should not reference them directly.

Another side-effect is the 'shifting' of the *long* or pointer fields. Although the overall layout is the same, the location offset of a *long* or pointer field within a data structure is different in 32-bit and 64-bit modes. When used in 32-bit mode, due to the presence of the 32-bit padding, the position of a *long* or pointer field will be shifted 4-bytes towards the end, compared to the same field when used in 64-bit mode.

Some of the data fields are of fixed 64-bit (e.g. *hssoff* in *zheader_type*) in both 32/64 modes. If a 32-bit program is to manipulate these 64-bit fields, it is recommended that the program should be compiled with the option -Ae (extended ANSI) or *-Ac* (K&R C mode), so as to enable the '*long long*' type for 64-bit integer.

The following sections describe the ZCOM data structures in **64-bit mode**. Hence the conditional padding fields are suppressed. If they are used in a 32-bit environment, the *long* and pointer fields will become 32-bit, and the 32-bit padding will be present.

NOTE

The following sections document the data structures used internally by the ACC ZCOM product to implement its functionality. These data structures are subject to change without notice. Programs that directly use these structures may be required to recompile their sources when a new version of the ZCOM subsystem is released (this will be indicated in the release notes).

ZCOM Header Structure

The ZCOM Header structure holds the system parameters and pointers to other areas of the ZCOM memory tables as well as other information that pertains to the whole ZCOM subsystem. The layout of the ZCOM Header structure *zheader_type* is shown in Table 3-1.

Table 3-1 ZCOM Header Structure

Field Name	Field Description	Field Type	Size (Bytes)
HLABEL	Header label	char [4]	4
HZREVC	ZCOM rev code	uns.short	2
HTREVC	TTGEN rev code	uns.short	2
HNAME1	Name of TTGEN source file	char [256]	256
HNAME2	Name of TTGEN object file	char [256]	256
HSNAME	System name (79 characters max)	char [80]	80
HSSOFF	System starting offset	uns.long long	8
HSSSIZE	Total system size in bytes	uns.int	4
HBPSIZE	Buffer pool size in bytes	uns.int	4
HNDMAX	Total number of node table entries	uns.short	2
HZLMAX	Total number of ZLUs	uns.short	2
HTMZLU	Number of terminal ZLUs	uns.short	2
HSPZLU	Number of program ZLUs	uns.short	2
HSPZL1	First program ZLU number	uns.short	2
HLTDSZE	Size of LTDATA in LT Table	uns.short	2
HLTMAX	Number of logical terminal tables	uns.short	2
HPTMAX	Number of physical terminal tables	uns.short	2

Table 3-1 ZCOM Header Structure (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
HNCARD	Number of interface tables	uns.short	2
HNRESP	Number of response records	uns.short	2
HNLTQL	Number of LTT queue labels	uns.short	2
HNLTSL	Number of LTT storage labels	uns.short	2
HPNTBL	Pointer to 1st Node table entry	pointer	8
HPZLU	Pointer to 1st Zlu table	pointer	8
HPTTBL	Pointer to 1st Logical terminal table	pointer	8
HPPTBL	Pointer to 1st Physical terminal table	pointer	8
HPIFTP	Pointer to 1st Interface table	pointer	
HPRESP	Pointer to 1st response record	pointer	8
HPQUES	Pointer to 1st Queue header	pointer	8
HPBFFR	Pointer to data buffer pool	pointer	8
HPQLIM	Default program queue limit (msgs)	uns.int	4
HUNACK	Default terminal unack limit (bytes)	uns.int	4
HMAXTX	Default pending TX limit (messages)	uns.int	4
HIPLIM	Default IFT port TX limit (msgs)	uns.int	4
HCTOTAL	Total number of DSC requests	uns.int	4
HCSTATE	DSC state If positive, it is the number of active DSC requests. If negative, DSC feature is suspended.	short	2
HCSWAIT	Number of DSC state waiters	uns.short	2
HRDATAQ	Remote data queue header	zqhd_type	56

Table 3-1 ZCOM Header Structure (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
HNDPID	ZNODE Process ID	int	4
HNDSIG	ZNODE required signal	int	4
HNIDLE	ZNODE idle timer value	uns.int	4
HNHIGH	ZNODE queue high-water mark	uns.int	4
HNLOW	ZNODE queue low-water mark	uns.int	4
HNWAIT	ZNODE high-water mark waiter	uns.short	2
HLCLND	Local node number	uns.short	2
HLTQTB	Queue label entries (16 bytes each)	struct [20]	20*16
HLTSTB	Storage label entries (16 bytes each)	struct[20]	20*16
DSC_ercvrs	Dyn. Reconfig. SEM receivers	pointer	8
ETPLIM	Default E1/T1 port transmit limit	uns.int	4
HOSTYPE	Type of operating system	int	4

HLABEL - ZMON identifier label

The identifier label contains the word 'ZMON' and provides compatibility with HP1000 version of ZCOM. The memory image file contains the word 'TTGE' in this position. The LDM overwrites this after the ZCOM subsystem has been initialized.

HZREVC - ZCOM revision code

TTGEN initializes this field with a 16-bit revision code, indicating the version of the table structures. This revision code is stored as four 4-bit digits. The first two digits represent the revision of a major release and the last two digits represent a minor release revision. Data structures and applications linked with a previous revision of the ZCOM software will be compatible if they have same major revision code. ZMON checks whether the revision code is acceptable. The different minor numbers indicate compatible revisions, such as bug fixes (as long as the major release revision code is the same).

HTREVC - TTGEN revision code

TTGEN inserts its integer revision code in this field in the memory image file. It is loaded into memory (unchanged) by ZMON and the LDM. This 16-bit revision code is stored in the same format as the ZCOM revision code (HZREVC). However, there is no validation of this field.

HNAME1 - Name of the TTGEN source file

Full path name of the source file used by TTGEN to generate the object file. This field is limited to 256 bytes and is null terminated.

HNAME2 - Name of TTGEN object file

Full path name of the TTGEN produced object file which was then used to startup (initialize) the ZCOM subsystem. This field is limited to 256 bytes and is null terminated.

HSNAME - ZCOM subsystem runtime instance name

Contains the contents of the *System-Name* parameter supplied in the TTGEN configuration file. This field is limited to 80 bytes and is null terminated.

HSSOFF - System starting offset

This pointer contains the starting physical kernel memory address of the ZCOM subsystem memory. It is initialized to zero by TTGEN. ZMON initializes it with the memory address while loading the memory tables into kernel memory. It is a 64-bit value when used in a 64-bit kernel.

HSSIZE - Total runtime system size in bytes

This is the total size in bytes of the ZCOM subsystem memory, including all tables, queue headers, and the buffer pool.

HBPSIZE - Buffer pool size in bytes

This is the size in bytes of the ZCOM buffer pool. It is defined by the *Buffer-Pool* parameter in the TTGEN configuration file.

HNDMAX - Number of node entries

This contains the maximum number of node entries which may exist in the node entries memory section. This value is defined by the *Node-Entry* keyword in the TTGEN configuration file. Each *Local-Node* and *Remote-Node* definition consumes one node table entry.

HZLMAX - Total number of ZLUs

Total number of terminal ZLUs and spare (i.e., program) ZLUs. That is, this is the combined values of the *Program*-ZLU and *Terminal*-ZLU parameters specified in the TTGEN configuration file.

HTMZLU - Number of terminal ZLUs

This is the number of ZLUs out of the total (HZLMAX) that are allocated to terminals. The terminal ZLU numbers are allocated from 1 up to the number specified in this field. The TTGEN *Terminal-ZLU* parameter is used to initialize this field.

HSPZLU - Number of program ZLUs

Total number of available program ZLUs. The TTGEN *Program-ZLU* parameter is used to initialize this field.

HSPZL1 - First program ZLU number

Number of the first ZLU not allocated to terminal ZLUs. This field is initialized to the value of the TTGEN parameter *Terminal-ZLU* + 1. It indicates where the program ZLUs begin within the ZLU Table. Note that program ZLUs are assigned from the range (*Terminal-ZLU*+1) to (*Terminal-ZLU* + *Program-ZLU*).

HLTDSZE - Logical terminal table LDATA buffer size

This specifies the size of the LDATA buffer in the logical terminal tables. This value is initialized from the *Logical-Size* parameter in the TTGEN configuration file. Set this parameter to 212 for backward compatibility with HP 1000 programs (so that the total size of Logical Terminal Table with extension is 512 bytes).

HLTMAX - Number of logical terminal tables

Contains the maximum number of Logical Terminal Tables (LTT) which may exist in the configuration. This value is initialized from the *Logical-Term* parameter in the TTGEN configuration file.

HPTMAX - Number of physical terminal tables

Contains the maximum number of Physical Terminal Tables (PTT) which may exist in the configuration. This value is initialized from the *Physical-Term* parameter in the TTGEN configuration file.

HNCARD - Number of Mux interface tables

Number of Interface Tables (IFTs) which may exist in the configuration. This value is initialized from the *Interface-Table* parameter in the TTGEN configuration file. Note that there is one Interface Table (IFT) per Mux card defined.

HNRESP - Number of response records

This field contains the number of response records in the ZCOM subsystem. This value is initialized from the *Program-ZLU* parameter in the TTGEN configuration file.

HNLTQL - Number of LTT queue labels

Number of entries in the queue label table, up to a maximum value of 20. It is initialized by TTGEN to the number of *Logical-Data* statements defined in the TTGEN configuration file. This value can be modified by calling the *zltqueue()* routine.

HNLTSL - Number of LTT storage labels

Number of entries in the storage label table, up to a maximum value of 20. It is initially set up by TTGEN and subsequently maintained by the LDM.

HPNTBL - Node Table pointer (znode_type *)

This is a pointer to the first entry in the Node Table. Note that all of the node entries are stored in sequential order within kernel memory.

HPZLU - ZLU table pointer (zlu_type *)

This is a pointer to the 1st entry in the ZLU table. Note that all of the ZLU table entries are stored in sequential order within kernel memory.

HPTTBL - Logical Terminal Table pointer (zltt_type *)

This is a pointer to the 1st Logical Terminal table. Note that all of the LTTs are stored in sequential order within kernel memory.

HPPTBL - Physical Terminal Table pointer (zptt_type *)

This is a pointer to the 1st physical terminal table. Note that all of the PTTs are stored in sequential order within kernel memory.

HPIFTP - Interface Table pointer (zift_type *)

This is a pointer to the first Interface Table (IFT). Note that all of the IFTs are stored in sequential order within kernel memory.

HPRESP - Pointer to the first response record (zrsp_type *)

This is a pointer to the first response record. Note that all of the response records are stored in sequential order within kernel memory.

HPQUES - Queue Header pointer (zqhd_type *)

This is a pointer to the first Queue Header. Note that all of the Queue Headers are stored in sequential order within kernel memory.

HPBFFR - Data buffer pointer (char *)

This is a pointer to the start of the data buffer pool. The data buffer pool is a block of contiguous kernel memory used for dynamically allocating data buffers for a variety of uses. The size of this block of memory is determined by the *Buffer-Pool* parameter from the TTGEN configuration file.

HPQLIM - Default program ZLU message queue limit

The default program ZLU message queue limit is the maximum number of messages that may be queued on each program ZLU. This value is initialized from the *Queue-Limit* parameter in the TTGEN configuration

file. The default limit may be overridden during a *zopen()* by specifying a non-zero value in the *limit* parameter. The program ZLU queue limit may be changed dynamically by calling the *zsetql()* routine.

HUNACK - Maximum number of unacknowledged Tx buffers pending on a terminal ZLU

Upper limit of the total size (in bytes) of all unacknowledged transmit messages to a terminal ZLU. If this limit has been reached, any further write requests to the terminal ZLU will be queued up in the terminal table, but will not processed by the driver immediately. The driver will resume processing of queued transmit requests when the Mux card acknowledges the completion of a previously issued transmit request for this terminal ZLU. This mechanism is used to prevent a few (or only one) terminal from consuming all of the transmit buffers on the Mux interface card. This field is initialized from the *Unack-Limit* parameter in the TTGEN configuration file. The unacknowledged queue limit may be changed dynamically by calling the *zsetql()* routine.

HMAXTX - Maximum number of Tx buffers pending on a terminal ZLU

Limit of transmit buffers pending on a terminal. When this limit is reached, the program doing write requests to the terminal will be suspended until some of the pending transmit requests are sent. Together with the limit imposed by *HUNACK*, this protects the ZCOM system buffers from excessive usage due to dead terminals (or other kind of device).

HIPLIM - Default IFT port TX limit (bytes)

This field contains the limit (in bytes) of all unacknowledged transmit messages to terminal ZLUs belonging to the same port of a 2/8-port Mux card. Each port on the card has approximately 24,480 bytes of buffer space available for both internal use and transmit requests. If the DAM issues a transmit request to the Mux card and the Mux card does not have enough free buffers to hold the request data, the Mux will reject the request with an "*out of buffers*" error. The DAM will retry the request after an unacknowledged transmit request completes. This field is used to minimize the number of retries (and thereby increase overall performance) by imposing a limit on the size of unacknowledged transmit requests. This value is configured through the *Port-Limit* parameter in the TTGEN configuration file and is used to initialize the individual port limits *iplimit[]* kept in the interface table (*zift_type*).

HCTOTAL - Total number of Dynamic System Configuration (DSC) requests

This field contains the number of DSC requests processed by the ZCOM subsystem since it was initialized and started up. It is incremented every time a DSC request is processed. This value is for information purposes only.

HCSTATE - Dynamic System Configuration (DSC) State

This field is used by the ZCOM subsystem as an indicator for controlling DSC functions. When the value is zero or positive, the DSC functions are enabled. The value represents the number of active DSC requests. When the value is negative, DSC functions are disabled. DSC requests are rejected until this field is set to zero. This field is initialized to zero by TTGEN.

HCSWAIT - Number of DSC state waiters

This field contains the number of processes suspended, waiting for the DSC state to be set to zero. That is, the processes are waiting for the DSC functions to be enabled. A process will be suspended if it issues a DSC request while the DSC functions are disabled or if it issues a DSC disable request while other DSC requests are in progress.

HRDATAQ - Remote data queue (ZNODE Queue)

This is the data queue for remote ZCOM requests. All ZCOM API requests to a remote node or deferred remote request responses will be added to this queue by the LDM. The remote node request daemon, ZNODE, reads and processes the request from this queue. This special queue is sometimes referred to as the *ZNODE queue* or *ZNODE data* queue.

HNDPID - ZNODE HP-UX process ID (PID)

This field contains the process ID (PID) of the local ZNODE daemon to use for signalling the arrival of data on the ZNODE queue. If this value is positive, the LDM will send a signal to this process ID whenever new data is placed in the queue. If this value is negative, a signal will be sent to the process group. The group ID used is the absolute value of this field. This field is initialized to zero by TTGEN and is set up by the ZNODE program when it starts up.

HNDSIG - ZNODE signal number

Contains the type of signal to use when informing ZNODE of the arrival of new data in the ZNODE queue. This field is initialized to zero by TTGEN and is set up by the ZNODE program when it starts up.

HNIDLE - ZNODE idle timer

This contains the time in seconds since ZNODE last issued a Node-Status Update ioctl request to the ZCOM subsystem (LDM). The LDM timer increments this field once every second and marks all nodes as DOWN when it exceeds the inactivity period (currently 20 seconds). This field is initialized to zero when the ZCOM runtime subsystem is started up.

HNHIGH, HNLOW - ZNODE queue high and low water marks (flow control)

These two fields are used by the LDM to control the flow of data between the ZNODE daemon and the LDM. When the number of pending messages on the ZNODE queue reaches the high-water mark, programs initiating remote requests are suspended to avoid excessive buffer usage. Suspended programs will be allowed to resume execution when the number of queued messages drops to or below the low-water mark.

A special case exists when both the high and low water marks are set to the same value. In this situation, whenever the number of pending messages on the ZNODE queue is at or above this mark, all remote requests are rejected with an immediate error return value of ZENBUSY (-1, ZNODE is busy).

Note that this mechanism is only applicable to programs initiating remote requests and does not limit the messages or responses generated as a result of ZCOM subsystem operation (e.g. a terminal ZLU returns a message to its receiver in a remote system).

These two values are defined by the Flow-Control keyword in the TTGEN configuration file. If not specified, they will default to a high-water mark value of the maximum queue limit and a low-water mark value of zero (i.e. the mechanism is disabled).

HNWAIT - ZNODE queue high-water mark waiter

This field is maintained and used by the LDM to keep track of the number of programs currently suspended due to the high/low water mark flow control mechanism. It is initialized to zero by TTGEN.

HLCLND - Local node number

The node number of the local ZCOM subsystem. If the *Node-Definition* section in the TTGEN configuration file is omitted, TTGEN defaults this field to one (1). Otherwise, it is initialized to the value of the *Local-Node* parameter. It is set up by TTGEN as the first node table entry that has zero ZLU number. There may be other numbers that also refer to the local node but HLCLND is the only one that is used to identify the local node on outbound messages.

HLTQTB - Queue label table

Each entry in the queue label table indicates the ownership of a data queue in a logical terminal table. Each entry has a structure of 16 bytes (*zqlb_type*) as shown in Table 3-2.

Table 3-2 Queue Label Table

Field Name	Field Description	Field Type	Size (bytes)
QLBGRP	Appl. grp. number (0 = globally assigned)	uns.short	2
QLBNAM	Queue label name	char	8
QLBQNB	Allocated queue number	uns.short	2
QLBRSV	<reserved>, unused	char	4

The table is initialized by TTGEN from the *Logical-Data* statements in the TTGEN configuration file. There is one entry in the Queue label table for each *Logical-Data* statement. Additional entries can be added to this table by calling the *zltqueue* routine (up to a maximum of 20 queue labels).

HLTSTB - Storage label table

The storage label entry indicates the ownership of an application data storage area in the logical terminal table extension area. Each entry is a structure of 16 bytes (*zslb_type*), as shown in Table 3-3.

Table 3-3 Storage Label Table

Field Name	Field Description	Field Type	Size (Bytes)
SLBGRP	Appl. grp. number (0 = globally assigned)	uns.short	2
SLBNAM	Storage label name	char	8
SLBSZE	Size of the application data in bytes	uns.short	2
SLBOFF	Offset from the start of the LTT to the application data in the extension area. Therefore, this value is the size of the LTT plus the offset of the application data from the start of the LTT extension area.	short	2
SLBRSV	<reserved>, unused	char	2

The table is initialized by TTGEN from the *Logical-Data* statements in the TTGEN configuration file. There is one entry in the Storage label table for each *Logical-Data* statement. Additional entries can be added to this table by calling the *zltqueue()* routine (up to a maximum of 20 storage labels).

DSC_ercvrs - Dynamic Reconfiguration System Event Receivers pointer (zshaddr_type *)

This field contains a pointer to a linked list of all applications that wish to be notified whenever a ZCOM subsystem dynamic reconfiguration system event message is generated (through the *zconfig* routine). This field is initialized by TTGEN to zero. It will only contain a value if at least one application has requested notification.

ETPLIM- Default E1/T1 IFT port TX limit (bytes)

This field contains the limit (in bytes) of all unacknowledged transmit messages to terminal ZLUs belonging to the same port and subchannel of an E1/T1 card. Each E1/T1 card has approximately 16 M-bytes of buffer space available for both internal use and transmit requests. If the DAM

issues a transmit request to the E1/T1 card and the card does not have enough free buffers to hold the request data, the Mux will reject the request with an "*out of buffers*" error. The DAM will retry the request after an unacknowledged transmit request completes. This field is used to minimize the number of retries (and thereby increase overall performance) by imposing a limit on the size of unacknowledged transmit requests. This value is configured through the *E1T1-Port-Limit* parameter in the TTGEN configuration file and is used to initialize the individual port and subchannel limit array *iplimit[]* kept in the interface table (*zift_type*).

HOSTYPE - Type of operating system

This field indicates the type of operating system that manipulates the data structures. The allowed values are 0, 32 (32-bit kernel) or 64 (64-bit kernel). It is first set up by TTGEN to 0. Then it is set to 32 or 64 by ZMON when it initializes the ZCOM kernel memory block.

Node Entries Table

The Node Entries table contains information about the remote ZCOM systems that communicate with the local ZCOM subsystem. Each *Local-Node* and *Remote-Node* definition in the TTGEN configuration file occupy one entry in the table. Each entry structure is 392 bytes long and has the format shown in Table 3-4.

Table 3-4 Node Entries Table

Field Name	Field Description	Field Type	Size (Bytes)
node_num	ZCOM system node number	uns.short	2
timeout	Default timeout value for this node	uns.short	2
flags	Node status and internal flags	uns.int	4
stats	Statistics for this node	struct	16
ndshzrvrs	List of node status event receivers	pointer	8
nhost	Number of hosts in the host array	uns.short	2
spare1	Reserved, not used	uns.short	2
spare2	Reserved, not used	int	4
host	Host link information	struct [4]	4*88

NODE_NUM - Node number

This field contains a unique node number that identifies the entry.

TIMEOUT - Default timeout value for this node

This field contains timeout in seconds for requests sent to this remote node. Most remote ZCOM requests (e.g. *ZOPEN*, *ZTIMR*, etc) require a definite response from the remote ZCOM subsystem. If the response is not received within the timeout period specified by this field, error ZENTOUT (-23) is returned to the application program. This field is initialized by TTGEN with the timeout value specified in a *Remote-Node* definition. This field is zero for a local node entry.

FLAGS - Node status and internal flags

This bit-field structure contains bits to maintain state and other internal information about this node. The contents of this field are as follows:

FLAGS.event - Set to 1 if there is at least one application that wants to be notified of a status change for the node.

FLAGS.valid - Set to 1 if this node entry in the table is valid; that is, currently defined and in use.

FLAGS.status - Set to 1 if this node is currently UP. The status bit indicates whether a remote node is usable. If this bit is not set, the LDM will reject all requests to that node with a ZENDOWN (-61) error. For a local node, the *status* bit indicates that the ZNODE daemon is running properly.

FLAGS.local - Set to 1 if this node entry in the table represents the local node.

STATS - Statistics for this node

This field is a data structure that contains some statistics on the data traffic for this node. The layout of this structure is shown in Table 3-5.

Table 3-5 STATS Data Structure

Field Name	Field Description	Field Type	Size (Bytes)
icount	Number of incoming messages	uns.int	4
ibytes	Number of incoming bytes	uns.int	4
ocount	Number of outgoing messages	uns.int	4
obytes	Number of outgoing bytes	uns.int	4

NDSHZRVRS - Pointer to a list of receivers for node status events.

This field contains a pointer to a linked list of applications to be notified whenever this node changes state. Whenever this node changes state from up-to-down or down-to-up, a node status system event message will

be sent to each program in this linked list. That is, those applications that have issued a *zevent_rcvr()* call to receive node status events. This field will be set to zero if there are no applications to be notified.

NHOST - The number of defined hosts in the host array.

This field contains the number of host link records defined in the HOST table below. This field is zero for a local node entry.

HOST - Host link information structure (array)

This array of structures contains information about each of the host links for this node. It can hold information on up to four host links. Each element of the array contains the fields for one host link as shown in Table 3-6.

Table 3-6 Host Link Information Structure

Field Name	Field Description	Field Type	Size (Bytes)
linkname	Host name or IP address for this link	char [64]	64
stats	Statistics for this link	struct	16
flags	Node status and flags	uns.int	4
spare3	Reserved, not used	int	4

LINKNAME - The name of the remote system

This field contains the host name or the IP address to be used in accessing the remote node via TCP/IP. If a host name is specified, it must be defined in the */etc/hosts* file.

STATS - Statistics for this host link

This field is a data structure that contains some statistics on the data traffic for this link. The layout of this structure is identical to that shown above. The data in this structure is maintained by the ZNODE daemon.

FLAGS - Host link status and flags

This bit-field data structure contains information on the status of a link. It is identical to the flags structure documented above. However only the *status* and *valid* bits are currently used. The *status* bit indicates whether

this link is currently up or not. If the link is down (*status* = 0), the ZNODE daemon will route remote requests on any secondary links that are up. The data in this structure is maintained by the ZNODE daemon.

ZLU Tables

The ZCOM logical unit table relates a terminal device or a program queue with a number or logical unit. The ZCOM logical units are called ZLUs to distinguish them from the HP-UX logical units (LUs).

Each ZLU entry structure contains 40 bytes of data.

The ZLU table is split into two logically distinct parts. The first part is the physical terminal area, and the second the dynamically assignable program ZLUs. The latter are used for program input queues. The ZLUs can be mapped so that a particular ZLU can point to any other ZLU.

Individual ZLU Entries

The individual ZLU data structure (*zlu_type*) layout is shown in Table 3-7.

Table 3-7 Individual ZLU Data Structure (*zlu_type*)

Field Name	Field Description	Field Type	Size (Bytes)
zcksum	ZLU checksum	uns.short	2
zlttype	ZLU type	uns.short	2
zpoint	ZLU pointer or mapped node nmbr	uns.short	2
zmpzlu	Mapped ZLU number	uns.short	2
zmpzcs	Mapped ZLU checksum	uns.short	2
zlname	ZLU Name (8 bytes)	char [8]	8
zowner	PID number of owner	uns.short	2
ztimer	Timeout interval (seconds)	short	2
zclock	Timeout clock	short	2
zlock	ZLU lock pointer	pointer	8
refcnt	Lock reference count	unns.int	4

Table 3-7 Individual ZLU Data Structure (*zlu_type*) (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
spare1	Reserved, not used	int	4

ZCKSUM - ZLU checksum

The checksum is used as an integrity check to ensure that a ZLU has not been altered while it is active. It is the 16-bit binary sum of bytes 3 to 18 of the ZLU entry (with any overflow discarded). It is calculated when the ZLU table entry is created.

ZLTYPE - Type of ZLU

The ZLU type values are shown in Table 3-8. The type value is also used to indicate whether the ZLU is defined.

Table 3-8 ZLU Type Values

Type	Description
0	ZLU is not defined
1	ZLU is local terminal (ZPOINT is a logical terminal number *)
2	ZLU is a local program (ZPOINT is a queue number)
3	ZLU is mapped (ZPOINT is a mapped ZLU node number)

* The logical terminal number is allocated by TTGEN from the TTGEN configuration file.

ZPOINT - ZLU pointer or mapped node number

The field is used to link the ZLU with the entity it represents. The meaning of the field is given in the table under ZLTYPE.

ZMPZLU - Mapped ZLU number

If the ZLU is type 3 (mapped ZLU), then this is the ZLU number (on the node as given by ZPOINT) onto which this ZLU is being mapped. If ZLU is not type 3, ZMPZLU is set to zero.

ZMPZCS - Mapped ZLU checksum

If the ZLU is not type 3, ZMPZCS is set to zero. Otherwise, this contains the checksum of the mapped ZLU.

ZLNAME - Name of ZLU

Name assigned to the ZLU by the creator of the ZLU. If the ZLU is the input queue to a program then this name is usually Z99999 - where 99999 is the program PID number. In this case the other 2 bytes are space filled. The program ZLU name can be assigned by the *zopen* call when the ZLU is allocated. The ZLU name can be found using *zinfo* given its node and ZLU number. The number of a ZLU can be found using *zname* given its name and node.

ZOWNER - PID number of owner

When the ZLU is created, the PID number of the creating program is inserted in this field. When the program closes the ZLU, this field is cleared. For system created ZLUs, this field is zero (e.g., terminal ZLUs assigned by TTGEN).

ZTIMER, ZCLOCK - ZLU timeout control

These two parameters act together as a timer clock. The timer value ZTIMER is set by an application program (via the time *parameter* in the *ztimr* routine). This field is moved into ZCLOCK and decremented by 1 every second. When ZCLOCK reaches zero, a timer message is added to the queue pointed to by the ZLU. ZCLOCK is then reset to ZTIMER and the timer continues. If there are already 10 or more messages on the queue, no timer message is queued. Timer processing stops when ZTIMER and ZCLOCK are reset to zero via the *ztimr* routine (a zero *time* parameter value).

ZLOCK - ZLU lock pointer

It is a pointer to a lock structure used within the kernel. The lock is used to provide exclusive access to the ZLU in a multi-processor environment. The locks are created one per ZLU by the ZCOM LDM driver during ZCOM subsystem startup.

REFCNT - Reference count

This field is to keep track of the number of active references to this ZLU. When the ZCOM subsystem processes an application request that refers to a ZLU, it increment this count. On completion, the count is decremented. On a zclos( ), the driver will wait until all active requests are cleared before closing and releases a ZLU.

Logical Terminal Tables

The Logical Terminal Table (LTT) pages contain the logical configuration information for each terminal ZLU in the ZCOM system. Each terminal table consists of the basic table (which is fixed for all terminals) and a table extension (which is of configurable size specified by *LOGICAL-SIZE* parameter in the TTGEN configuration file).

The basic table contains common information for all terminal ZLUs. It is logically divided into two sections; an area reserved for use by the ZCOM subsystem and a user area that may be manipulated by application programs (via the *zltup* routine).

The table extension may be used to store any application defined data structure and can be manipulated by the application program that allocates it (via the *zltstore* routine).

The layout of a Logical Terminal Table (*zltt_type*) basic table is shown in Table 3-9 and Table 3-10. Note that the section reserved for use by the ZCOM system is immediately followed by the user area.

Table 3-9 Logical Terminal Table (Reserved Area)

Field Name	Field Description	Field Type	Size (Bytes)
LTZLU	Owning terminal ZLU number	uns.short	2
LTZCS	Owning terminal ZLU checksum	uns.short	2
LTMUX	MUX i/f number for this terminal	uns.short	2
LTPORT	Port number within Mux interface	uns.char	1
LTTERM	Terminal number within port	uns.char	1
LTPTERM	Pointer to physical terminal table	pointer	8
LTGZLK	ZLU of next terminal in group	uns.short	2
LTDEVC	Device type of terminal	uns.short	2
LTTYPE	Logical type of terminal	uns.short	2
LTFLAG	System status flags	uns.short	2

Table 3-9 Logical Terminal Table (Reserved Area) (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
LTZMXP	ZCOM address of multiplexing program	struct	6
LTZPRCVR	ZCOM addr. of indirect primary receiver	struct	6
SPARE1	Reserved, not used	int	4
LTZSHRCVRS	Pointer to list of indirect shared receivers	pointer	8
LTHLDQ	Sub-packet holding queue (0)	struct	56
LTHDQN	Data holding queues (1~4)	struct [4]	4*56

This is followed by LTUSER - the user maintainable area - which consists of the fields shown in Table 3-10.

Table 3-10 Logical Terminal Table (User Maintainable Area)

Field Name	Field Description	Field Type	Size (Bytes)
LTADDR	Logical terminal address	uns.short[10]	10
LTSTAT	Logical status	uns.short	2
LTCWCT	Control write counter	uns.int	4
LTTXCT	Transmit counter	uns.int	4
LTRXCT	Receive counter	uns.int	4
LTERCT	Error counter	uns.int	4
LTAPNO	System application number	uns.short	2
LTINST	Terminal institution number	uns.short	2
LTBRCH	Terminal branch number	uns.short	2
LTWORK	Terminal workstation number	uns.short	2
LTAREA	Terminal area number	uns.short	2
LTNAME	Terminal name (30 bytes)	char [30]	30

Table 3-10 Logical Terminal Table (User Maintainable Area)

Field Name	Field Description	Field Type	Size (Bytes)
SPARE2	Reserved, not used	int	4

LTZLU - Terminal ZLU number

This is the ZLU number of the owning terminal.

LTZCS - Owning ZLU checksum

This is the ZLU checksum of the owning terminal.

LTMUX - Mux interface number for this terminal

This is the Mux interface number that this terminal is configured on.

LTPORT- Port number within the Mux card

This is a one byte field that indicates which port this terminal is attached to within the Mux card.

LTTERM - Terminal number within port

This is a one byte field that indicates which terminal within a port this terminal is configured on. This field has been replaced by the *ltmxtm* field. This field continues to be valid for 2/8-channel MUX cards for backwards compatibility reasons.

LTPTERM - Pointer to the physical terminal table (zptt_type *)

It contains the memory address of the related physical terminal table. This field is used as a linkage to find physical terminal information.

LTGZLK - ZLU of next terminal in group

Contains the circular linkage of ZLUs linked to the same physical terminal. The linkage is to allow de-multiplexing of inbound messages. For a normal 1:1 terminal, the entry will contain its own ZLU number. Note that there is no checksum associated with this ZLU.

LTDEVC - Terminal device type

Unique identification of the actual terminal connected to the system. Two different terminals can have the same screen format and physical protocol, but the device type may not necessarily be the same; it relates to the symbolic device type specified in the TTGEN configuration file.

LTTYPE - Logical terminal type

The logical terminal type is used to describe the message format expected from the terminal. For screen-type terminals this will identify the terminal as belonging to a class of terminals with the same screen formatting attributes and message delimiters (e.g., IBM.3278 or NCR.501).

LTFLAG - System status flags

Currently the system status flags are used to indicate whether the outbound multiplexing and the inbound multiplexing are enabled.

Figure 3-2 System Status flags

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OM	IM														

OM - Outward bound multiplexing, ZCOM_LTFLAG_OMX bit
IM - Inbound multiplexing, ZCOM_LTFLAG_IMX bit

Both are set by TTGEN according to the device type specified. These flags may be modified dynamically by use of the *zltmx* routine call.

LTZMXP - ZCOM address of multiplexing program

This field contains the node, ZLU number, and checksum of the outbound multiplexing program. If the ZCOM_LTFLAG_OMX bit in *ltflag* is set, the responsible multiplexing program should set up this field with its primary ZLU (via the *zset_rcvr* routine, *mode* 2). All out-going messages will then be queued to its primary ZLU.

LTZPRCVR - ZCOM address of primary indirect receiver

This field contains the node, ZLU number, and checksum of the program to receive data messages from an inbound multiplexed terminal. If the ZCOM_LTFLAG_IMX bit in *ltflag* is set, the system sets up this field on a *zset_rcvr* (*mode 0*) call. All incoming messages will be received by the inbound multiplexing program, which will then route the data to the receiver specified in this field.

LTZSHRCVRS - Pointer to a linked list of indirect shared receivers

This field contains a pointer to a linked list of shared receivers that wish to receive data messages from an inbound multiplexed terminal. Each entry in the linked list represents one program. If the ZCOM_LTFLAG_IMX bit in *ltflag* is set, the system sets up this field on a *zset_rcvr* (*mode 0*) call. All incoming messages will be received by the inbound multiplexing program, which will then route the data to each of the receivers specified in this linked list.

LTHLDQ - Sub-packet holding queue

This queue is a holding queue for partial messages to an inbound multiplexed terminal.

LTDHQN - Data holding queues

For use by an application program to store temporary data for each terminal. Only the queue header is stored in these fields, as the data itself is stored in the ZCOM subsystem buffers.

LTUSER - User-maintainable area

The fields in the user-maintainable area may be modified by the *zltup* call. The area consists of the following fields (up to the end of the terminal table):

LTADDR - Logical terminal address

Holds the logical terminal addressing information needed to recognize the correct logical terminal in a group of multiplexed terminals. This is a 10-byte field, however the multiplexing program may use these bytes according to the specific requirements of the protocol.

LTSTAT - Logical status

Contains the status of the logical terminal. This entry is maintained by a multiplexing program and will remain unaltered for a normal 1:1 terminal.

Figure 3-3 Logical Status Format LTSTAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	AC	UP													

EN - logical terminal enabled (ZCOM_LTSTAT_ENB)
AC - logical terminal activated (ZCOM_LTSTAT_ACT)
UP - logical terminal up (ZCOM_LTSTAT_UP)

LTCWCT, LTTXCT, LTRXCT, LTERCT - Message and error counters

Allow message counters for control writes, transmit data, receive data, and error events to be maintained by the multiplexing application program, for multiplexed logical terminals.

LTAPNO - System application number

This field is initialized by TTGEN based on the value specified in the *Term* or *Lterm* statement in the TTGEN configuration file. This field is not used by the ZCOM subsystem. It allows applications to recognize the terminals which are assigned to it. Each ZCOM application which is a direct receiver of terminal messages should access only those terminals with its allocated application number.

LTINST, LTBRCH, LTWORK, LTAREA - Terminal logical ID

These logical terminal ID fields describe the owner and location of the terminal at an application program level. At this level the physical parameters are not relevant. Because these fields are at an application level, their contents are largely dependent on the particular application.

LTNAME - Terminal name

This is a 30-byte ASCII description of the terminal. It is used by the ZCOM utility programs as well as application programs. The field is initialized from the *desc* parameter of the *Term* or *Lterm* statements in the TTGEN configuration file.

The logical terminal table extension follows the basic table, and is used exclusively by the application programs. The size of this area is specified in the HLTDSZE field in the ZCOM header table (*zheader_type*).

Physical Terminal Tables

The physical terminal table (*zptt_type*) contains information used by the LDM and DAM to initialize and control a physical terminal (or other device). The PTT consists of a system area, which is reserved for use by the ZCOM subsystem, and a user area which may be manipulated by application programs. The user modifiable portion of the PTT immediately follows the reserved area. The layout of the PTT is shown in Table 3-11 and Table 3-12.

Table 3-11 Physical Terminal Tables (System Area)

Field Name	Field Description	Field Type	Size (Bytes)
PTZLU	Owning terminal ZLU number	uns.short	2
PTZCS	Owning terminal ZLU checksum	uns.short	2
PTMUX	MUX number for this terminal	uns.short	2
PTPORT	Port number within interface	uns.char	1
PTTERM	Terminal number within port	uns.char	1
PTMX_TERM	Terminal number in card	uns.short	2
SUBCH	Subchannel number on port	uns.short	2
PTZPRCVR	ZCOM address of primary receiver	struct	6
PTZPCTL_RCVR	ZCOM address of primary control receiver	struct	6
PTZSHRCVRS	Pointer to list of shared receivers	pointer	8
PTZSHCTLRCVRS	Pointer to list of shared control receivers	pointer	8
PTLIST	Pointer to next PTT of same port	pointer	8
PTTXLX	Pointer to next PTT on express list	pointer	8
PTTXQX	Express transmit buffer queue	struct	56

Table 3-11 Physical Terminal Tables (System Area) (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
PTTXLA	Pointer to next PTT on high priority list	pointer	8
PTTXQA	High priority transmit buffer queue	struct	56
PTTXLB	Pointer to next PTT on low priority list	pointer	8
PTTXQB	Low priority transmit buffer queue	struct	56
PTTXQC	Unacknowledged transmit buffer queue	struct	56
PENDG_TXREQ	Next request for $DATA	struct	56
PTPZLK	ZLU of next PTT on this port	uns.short	2
PTDRST	Driver terminal status	uns.short	2
PTFWST	Firmware terminal status	uns.short	2
PTTYPE	Physical type of terminal	uns.short	2
SCFG_LEN	Length of special configuration	uns.short	2
SPEC_CFG	Special protocol configuration	uns.char[6]	6

Table 3-12 Physical Terminal Tables (User Maintainable Area PTUSER)

Field Name	Field Description	Field Type	Size (Bytes)
PTPOLL	Terminal poll address	uns.short	2
PTSLCT	Terminal select address	uns.short	2
PTOPTN	Protocol options	uns.short	2
PTVCNO	X.25 logical channel number	uns.short	2

Table 3-12 Physical Terminal Tables (User Maintainable Area PTUSER) (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
PTCNFG	Special configuration	uns.char [8]	8
PTCWCT	Control write counter	uns.int	4
PTTXCT	Transmit message counter	uns.int	4
PTRXCT	Receive message counter	uns.int	4
PTERCT	Error counter	uns.int	4

PTZLU - Owning terminal ZLU number

PTZCS - Owning terminal ZLU checksum

The terminal ZLU number points back to the ZLU table entry for the owner of the terminal. The ZLU contains the checksum for the ZLU as well as its number. It is used by the driver to derive the source ZLU for all messages received for this physical terminal. This parameter is initially setup by TTGEN in the memory image file. These fields are maintained by the ZCOM subsystem when ZLU and terminal linkage are modified.

PTMUX - Mux number of terminal

The Mux interface number of the terminal. This is set initially through the TTGEN configuration file and is subsequently maintainable.

PTPORT - Terminal port number

The port number of the terminal on the Mux interface. This is set initially through the TTGEN configuration file and is subsequently maintainable.

PTTERM - Terminal number within port

The terminal number on the Mux interface card within the port. This is set initially through the TTGEN configuration file and is subsequently maintainable. This field has been replaced by the *ptmx_term* field. It remains valid for 2/8-channel cards for backwards compatibility reasons.

PTMX_TERM - Terminal number within port or card

This field contains the firmware terminal number this terminal ZLU is associated with. The firmware manages terminals using a terminal number while the ZCOM subsystem manages terminals using ZLUs on the host. On 2/8-channel cards, the terminal number must be unique for a given port. On E1/T1 cards, the terminal number is unique per card.

SUBCH - Subchannel number within port

This field contains the subchannel this terminal ZLU is configured on. If the card does not support subchannels, this field is set to zero.

PTZPRCVR - ZCOM address of primary receiver

Node number, ZLU, and checksum used as the destination address for normal messages and unsolicited status messages received for this physical terminal. This field is set by the *zset_rcvr* call (mode 0 or mode 1 if multiplexed terminal).

PTZPCTL_RCVR - ZCOM address of primary control receiver

This is the node number, ZLU, and checksum used as the destination address for any control input messages received for this terminal. This field is set by a *zset_rcvr* mode 3 call. The control input messages are distinguished by the protocol modules on the MUX, by setting bit 1 of the input message tag byte.

PTZSHRCVRS - Pointer to a linked list of shared receivers

This field contains a pointer to a linked list of programs that wish to receive normal messages and unsolicited status messages from this physical terminal. Each entry in the linked list contains the node number, ZLU, and checksum of the application which will be used as the destination address for normal messages and unsolicited status messages received from this physical terminal. This field is set by the *zset_rcvr* call (mode 0 or mode 1 if multiplexed terminal). If there are no shared receivers, this field will contain a zero. When a data or status message arrives, it will be delivered to each application in the linked list (as well any primary receiver).

PTZSHCTLRCVRS - Pointer to a linked list of shared control receivers

This field contains a pointer to a linked list of programs that wish to receive control messages from this physical terminal. Each entry in the linked list contains the node number, ZLU, and checksum used as the destination address for any control input messages received for this terminal. This field is set by a *zset_rcvr* mode 3 call. The control input messages are distinguished by the protocol modules on the MUX, by

setting bit 1 of the input message tag byte. If there are no shared control receivers, this field will contain a zero. When a control message arrives, it will be delivered to each application in the linked list (as well an any primary control receiver).

PTLIST - Pointer to next PTT on the same port

Contains a pointer to the physical terminal table of the next terminal on the same Mux interface card and port number as the current terminal. These pointers form a circular list of terminals on the same port. Field PTPZLK forms the same circular list, but contains the ZLU number.

PTTXLX - Pointer to next PTT on the express list linkage

This linkage (containing the physical terminal table address) is used to link together physical terminals on the Mux which have messages queued on the express transmit queue. This field is NULL if the terminal is not on the list. It is initialized to NULL by TTGEN and is maintained by the LDM and DAM.

PTTXQX - Express transmit buffer queue

This queue contains the express transmit requests waiting to be sent. It also contains the terminal configuration (*zcntl*) requests. When a message has been transferred to the card, the buffers are moved to the unacknowledged transmit buffer queue. Once the buffer has been acknowledged as sent or flushed by the card it is removed from the unacknowledged queue and returned to the sending program (if requested by the program).

The express transmit queue is designed for passing critical control (protocol dependent) data to the terminal. In general, none of the normal queue limit mechanisms apply to requests on this queue. Normal user data should **NEVER** be placed on this queue. An example of appropriate requests would be an X.25 Reset request or X.25 Expedited data. Requests on this queue are always processed first, regardless of any transmit limits that may have been reached.

PTTXLA - Pointer to next PTT on the high priority list linkage

This linkage (containing the physical terminal table address) is used to link together physical terminals on the Mux which have messages queued on their high priority transmit queue. This field is NULL if the terminal is not on the list. It is initialized to NULL by TTGEN and is maintained by the LDM and DAM.

PTTXQA - High priority transmit buffer queue

This queue contains the high priority transmit requests waiting to be sent. It also contains the terminal configuration (*zcntl*) requests. The data on this queue is processed only after all express transmit data has been processed. When a message has been transferred to the card, the buffers are moved to the unacknowledged transmit buffer queue. Once the buffer has been acknowledged as sent or flushed by the card it is removed from the unacknowledged queue and returned to the sending program (if requested by the program). Note that requests on this queue are subject to the unacknowledged and port transmit limits.

PTTXLB - Pointer to next PTT on low priority list linkage

As for PTTXLA above, except that the linkage pertains to the low priority transmit queue.

PTTXQB - Low priority transmit buffer queue

The low priority queue is operated in a similar manner to the high priority queue. Data on this queue is processed only after all express and high priority data has been processed. Note that requests on this queue are subject to the unacknowledged and port transmit limits.

PTTXQC - Unacknowledged transmit buffer queue

This queue contains the buffers from PTTXQA/B after they have been transferred to the interface card, and until they have been acknowledged as sent (or aborted) by the Mux interface card.

PENDG_TXREQ - Next request of $DATA in unack'd transmit queue

This field is used internally by the ZCOM subsystem.

PTPZLK - ZLU of next PTT on this port

Contains the ZLU number of the next terminal which is on the same Mux interface and port as this terminal. These ZLU numbers form a circular list of terminals on the same port. Field PTLIST forms the same circular list, but contains the physical terminal table memory address.

PTDRST - Terminal status

The terminal status is maintained by the driver in this field. The status change requests from application programs are updated in this field after they are successfully passed to the interface card. The driver uses this value to reset the terminal status on the Mux interface after a powerfail or reset.

The content of this field is as follows:

Figure 3-4 Terminal Status Field PTDRST

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	AC	UP	AV												

IN - Terminal has been initialized (ZCOM_PTDRST_INT)
EN - Terminal is enabled (ZCOM_PTDRST_ENB)
AC - Terminal is active (ZCOM_PTDRST_ACT)
AV - Terminal is available for use

The IN bit indicates whether the terminal is considered as part of the current configuration. It is initialized by TTGEN and maintained by the driver. The EN and AC bits represent the current status of the terminal. They are initialized by TTGEN and maintained by the driver. They are reflected in the opposite sense by DS and IA on the interface card and in PTFWST.

When the ZCOM subsystem starts up, the driver sets the AV bit if the terminal is successfully initialized.

PTFWST - Firmware terminal status

This word contains the TMSTAT and TMSTT2 bytes from the Mux (via $STDT buffer)

Figure 3-5 Firmware Terminal Status PTFWST

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DS	SP	IA	GP	DN	DC	GA	GM								

DS - Terminal is disabled
SP - Terminal is on slow poll list
IA - Terminal is inactive (no poll)
GP - Terminal is group polled
DN - Terminal is down
DC - Terminal downcounter expired
GA - Group poll active
GM - Group poll master

The status in bits 7-0 is copied from the $STDT buffer. These bits represent the field TMSTT2 from the interface terminal table.

PTTYPE - Terminal type (physical)

Protocol module terminal type. It is initially setup by TTGEN in the memory image file but is subsequently maintainable.

SCFG_LEN - Length of the special protocol configuration bytes.

This field contains the length of any protocol specific configuration data contained in the *spec_cfg[]* field.

SPEC_CFG - Special protocol configuration data

This field contains any protocol specific configuration data that might be needed. The details on its contents and how to configure this field is documented in the protocol manuals.

PTUSER - User-maintainable area

This area extends to the end of the physical terminal table. It defines the area that may be modified by a *zptup* call, and contains all the following fields:

PTPOLL - Terminal poll addresses

PTSLCT - Terminal select addresses

These words usually contain the poll and select addresses specified for the terminal in the TTGEN configuration file. For some protocols however, they contain other configuration information. Specific content for these words is documented in protocol accessory manuals.

PTOPTN - Protocol options

Used to define options to the protocol module on the Mux interface card. Only bits 0 - 7 are relevant to the Mux. Bits 8 - 15 may contain other (host level) options. The definition of the bits depends on the protocol (PTTYPE) type.

PTVCNO - X.25 logical channel number

Used to contain status information - by *x25cn* & *zx25d* for X.25 terminals.

PTCNFG - Special configuration

For X.25, contains a call number.

PTCWCT - Terminal control write counter

This counter is incremented by the driver every time a control write buffer for this terminal is transferred to the Mux interface card.

PTTXCT - Terminal transmit message counter

This counter is incremented by the driver every time a transmit buffer for this terminal is transferred to the Mux interface card.

PTRXCT - Terminal receive message counter

This counter is incremented by the driver every time a receive message is transferred from the Mux interface card for this terminal.

PTERCT - Terminal error message counter

This counter is updated by the driver after each $STDT buffer has been transferred from the Mux interface card. The $STDT buffer contains the number of errors that have occurred for a terminal since the last $STDT buffer was sent to the driver.

Interface Table

The interface table shown in Table 3-13 is a structure that has one instance for each Mux interface card defined in the TTGEN configuration file. It is primarily used by the DAM for controlling the backplane interaction with the Mux interface card. (*zift_type*)

Table 3-13 Interface Table

Field Name	Field Description	Field Type	Size (Bytes)
istime	Time of ZMON control of card	int	4
bc1_addr	Level 1 Bus converter address	uns. char	1
bc_addr	Bus converter address for this card	uns.char	1
card_addr	Card address of this interface	uns.short	2
inhpa	Interface HPA value from TTGEN	long	8
iftype	Interface card type	uns.short	2
ifstat	Interface card status	uns.short	2
ischdl	Scheduler event flags	uns.short	2
itcntr	Interface terminal count	uns.short	2
ifrtry	Reset retry count	uns.short	2
irserr	Restart error count	uns.short	2
ifstcn	Status update request count	uns.short	2
intfno	Interface card number	uns.short	2
imaxports	Maximum # of ports per card	uns.short	2
imaxsubc_port	Max. # of subchannels per port	uns.short	2
imaxsubc_card	Max. # of subchannels per card	uns.short	2

Table 3-13 Interface Table (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
imaxterms	Max. # of terminals per card	uns.short	2
ifname	Firmware download file name	char [256]	256
iffldt	Download file link time	uns.int	4
iffnmn	Download file module name	char [12]	12
ifrudt	ROM label: ROM update time	uns.int	4
ifrmnm	ROM label: module name	char [12]	12
ifrrev	ROM label: revision code	char [8]	8
ifwinfo	Firmware run-time information word	uns.short	2
ifboff	Buffer pointer adjusted offset	uns.short	2
spare1	Reserved, not used	int	4
intfcq	Interface control queue	struct	56
ifdmaq	DMA receive buffer queue	struct	56
txcx_head	Express queue terminal lists	Pointer	8
txca_head	High-priority queue terminal lists	Pointer	8
txcb_head	Low-priority queue terminal lists	Pointer	8
itxlstx	Express terminal list headers	Pointer	8
itxlsta	High priority terminal list headers	Pointer	8
itxlstb	Low priority terminal list headers	Pointer	8
iplimit	Port unack TX limit (in bytes)	Pointer	8
ipbytes	Port unack TX size (in bytes)	Pointer	8
ipstat	Port status/Subchannel status	Pointer	8
ipcntr	Pointer to terminal count per port	Pointer	8

Table 3-13 Interface Table (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
iportn	Pointer to port configuration bytes	Pointer	8
ifplook	PTT lookup table pointer	Pointer	8
ifcmdbp	Current backplane command buffer pointer	Pointer	8
ilock	Pointer to IFT lock	Pointer	8
ihpap	Pointer to HPA entry	Pointer	8
card	Card type specific information	Union of: bsp mmp	 3616 18592

In the following definitions of the different fields, there are references to $PORT, $FIRQ and $STDT, transactions. For more information on these transactions, refer to the backplane protocol descriptions.

ISTIME - Time of last ZMON restart/shutdown this card.

This field contains the time (seconds since the Epoch) when ZMON last processed a restart or shutdown request for this card. It is used internally by ZMON to detect and ignore multiple restart/shutdown requests.

BC1_ADDR - First level bus converter address for this MUX interface card

For HP9000 Series K systems the address is of the form bc1/bc/s. This field refers to the first level bus converter address indicated as bc1. For other systems this field is initialized to zero.

BC_ADDR - Bus converter address for this MUX interface card

This field contains the I/O bus converter address representing the I/O bus that the MUX interface card is installed in. On systems that contain only one I/O bus (S8x7, I30, H50, etc.), this field is initialized to zero.

CARD_ADDR - Card address for this MUX interface card

This field contains the I/O card address representing the I/O slot number that the MUX interface card is installed in. It is initialized by TTGEN from the slot value specified in the *MUX* statement of the TTGEN configuration file.

INHPA - Interface HPA value (from TTGEN)

The HPA value is built as follows:

Table 3-14 Interface HPA Value Format INHPA

63-32	31-28	27-18	17-14	13-12	11	10-6	5-2	1-0
All 1's (111...)	1111	Bus Address	Slot	Module	Page	Register Set	Reg No.	00

For the Mux interface in an HP-PB standard backplane, the card address is 4 x slot number. In the TTGEN configuration file, the card address is given as BUS:SLOT (e.g., 0:48), or BUS:BUS:SLOT. Here the 10-bit Bus Address field is translated from the 1's complement of BUS (i.e., 1023-BUS). The Slot field is specified by SLOT/4. The Module, Page, Register Set, and Reg No. fields are set to 0. For the example of BUS:SLOT=0:48, the HPA is 0xFFFB0000. For EISA standard backplanes the address is the slot number.

IFTYPE - Interface card type

Specified in TTGEN through the interface definition (the keyword MUX) with the following results:

```
IFTYPE   = ZCOM_ITYPE_Z7200A(6)  - Z7200A HP-PB ACC 8-channel Mux
IFTYPE   = ZCOM_ITYPE_Z7350A(7)  - Z7350A HP-PB ACC 2-channel Mux
IFTYPE   = ZCOM_ITYPE_Z7400A(8)  - Z7400A EISA  ACC 8-channel Mux
IFTYPE   = ZCOM_ITYPE_Z7300A(9)  - Z7300A HP-PB ACC 4-port E1/T1
IFTYPE     = ZCOM_ITYPE_Z7330A(10) - Z7330A PCI  ACC 4-port E1/T1
IFTYPE   = ZCOM_ITYPE_Z7340A(11) - Z7340A PCI   ACC 8-channel Mux
```

This field allows for the future support of extra interface types.

IFSTAT - interface card status

This word indicates the current status of the Mux card. It is updated by the driver. Bit 15 indicates whether the card is usable (when set) or not (when cleared).

Table 3-15 Interface Card Status IFSTAT

Symbol	Value (hex)	Meaning
ZCOM_IFSTAT_BOOT	0x0000	Driver and interface not initialized
ZCOM_IFSTAT_RST	0x8001	Being reset
ZCOM_IFSTAT_TMO	0x8002	Timed out
ZCOM_IFSTAT_PWF	0x8003	Power failed
ZCOM_IFSTAT_DWN	0x0004	Card down (due to serious error)
ZCOM_IFSTAT_ACT	0x8005	Up and active
ZCOM_IFSTAT_FWF	0x8006	Firmware failure
ZCOM_IFSTAT_OPR	0x8007	Operator reset
ZCOM_IFSTAT_DMI	0x8008	DMA interrupt error
ZCOM_IFSTAT_CDD	0x8009	Card dead, no activity
ZCOM_IFSTAT_NOC	0x000A	Card absent
ZCOM_IFSTAT_FER	0x800B	Hardware fatal error
ZCOM_IFSTAT_DSB	0x000C	Card disabled
ZCOM_IFSTAT_INC	0x000D	Card startup incomplete
ZCOM_IFSTAT_SUSP	0x000E	Card being suspended (OLAR)

When the card status is down (i.e., ZCOM_IFSTAT_DWN), it can only be resumed through a control request to the driver which will reset the driver and the interface card (via "*zmon restart*").

ISCHDL - Scheduler event flags

The scheduler uses this word to store the event flags that control the state of the backplane.

Figure 3-6 Scheduler Event Flags ISCHDL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AC	ST	NB	PA												

AC - Driver currently active
ST - Card has sent status transfer request ($STDT)
NB - Driver cannot get any free system buffers
PA - Pause after current event

ITCNTR - Interface terminal counter

The terminal counter is the actual number of terminals defined on this Mux interface card.

IFRTRY - Reset retry count

This is used to count the number of times the Mux interface has been reset due to a backplane timeout, or firmware failure. It is incremented by ZMON each time the Mux card is reset by a request from the DAM. If it reaches an unsatisfactory level, the Mux card will be disabled (set down) by ZMON.

The driver will decrement the retry count if it is not zero after every 30 successful status $STDT replies (i.e., every 5 minutes). If an unstable Mux interface fails and restarts too often, the ZCOM subsystem will mark it as “down” and stop using it.

IRSERR - Restart error count

This field contains the count of the number of errors which occurred during the last restart or enable. These errors are due to $PORT or $TERM commands and will be separately reported to ZMON. This field is kept for information purposes only.

IFSTCN - Status update request count

This counts the number of $STDT replies until IFRTRY will be decremented.

INTFNO - Interface card number

This is the interface number that was specified in the Mux statement of the TTGEN configuration file.

IMAXPORTS - Maximum number of ports on card.

This field contains the maximum number of ports supported by this specific ACC interface card.

IMAXSUBC_PORT - Maximum number of subchannels per port.

This is the maximum number of subchannels supported on each port of this ACC interface card. The minimum value of this field is 1 (e.g. 2/8-channel cards).

IMAXSUBC_CARD - Maximum number of subchannels on card.

This field contains the number of subchannels supported by this card. For a non-E1/T1 card, this value should be the same as the *imaxports* field.

IMAXTERMS - Maximum number of terminals allowed on card.

This field contains the maximum number of terminal ZLUs that may be configured on this card.

IFNAME - Firmware download file name

This contains the full path name of the firmware file to be downloaded into this Mux card whenever the ZCOM subsystem is started, the Mux card is reset, or is recovering from a powerfail or firmware failure. It is initialized from the information supplied in the Mux statement of the TTGEN configuration file. This field is a maximum of 256 bytes including the null terminator byte.

IFFLDT, IFFMNM - Download file link time and module name

These two fields contain the time (since Epoch) when the download firmware file is linked (by *zlink* program) and its module name. These are set up by *zlink* and stored in the download file. When ZMON loads the file to the Mux interface card, it extracts them from the file and puts them into these two fields. The MX command in *zmntr* displays these fields.

IFRUDT, IFRMNM, IFRREV - ROM update time, module name and revision code

The ZCOM subsystem ROM contains its module name, revision code and update time in byte locations 40~5D hex. ZMON reads these bytes from the Mux interface card when it loads the firmware file, and stores them in these fields. The MX command in *zmntr* displays these fields. The time is translated from the ROM ASCII time stamp and stored in Epoch format.

IFWINFO - Firmware run-time information word

This is a 16-bit field refreshed by the DAM once every 10 seconds with the runtime information word from the Mux firmware. It contains the average percentage activity of the Mux interface for the previous 10 seconds and the type code of the Mux front panel connected.

IFBOFF - Buffer pointer adjusted offset

Contained with the interface table are buffers that are used for DMA transfers between the driver and the Mux card. These buffers are a multiple of 64 bytes and must be aligned on a 64-byte address boundary. This field contains the byte offset needed to cause the buffers to start on a 64-byte boundary. TTGEN initializes this field with the byte offset value.

INTFCQ - Interface control queue (zqhd_type)

This queue contains pending port requests from application programs. The operation of this queue is described in the section on the DAM. This format of the queue header structure (*zqhd_type*) is described later in this chapter.

IFDMAQ - DMA receive buffer queue (zqhd_type)

Holds the current receive buffer during an $RXDT DMA transfer. This format of the queue header structure (*zqhd_type*) is described later in this chapter.

TXCX_HEAD - Pointer to head of the express queue linked lists

TXCA_HEAD - Pointer to head of the high-priority queue linked lists

TXCB_HEAD - Pointer to head of the low-priority queue linked lists

ITXLSTX - Pointer to express terminal list array (in union)

ITXLSTA - Pointer to high-priority terminal list array (in union)

ITXLSTB - Pointer to low-priority terminal list array (in union)

IPLIMIT - Pointer to port/subchannel unack Tx limit array (in union)

IPBYTES - Pointer to port/subchannel unack Tx size array (in union)

IPSTAT - Pointer to port/subchannel status array (in union)

IPCNTR - Pointer to # of terminals per port/subchannel array (in union)

IPORTN - Pointer to port configuration array (in union)

IFPLOOK - Pointer to PTT interrupt vector array.

IFCMDBP - Pointer to current backplane command buffer (in union)

These fields are used internally by the ZCOM subsystem.

CARD - Interface card specific data structures (union)

This field is a union which contains two structures. These structures hold information that are specific to the type of interface card associated with this IFT entry. The structure *card.mmp* is used if the IFT is associated with an 8-port PCI or E1/T1 interface card. Otherwise, the *card.bsp* structure is used for 2/8-channel NIO and EISA interface cards.

CARD.MMP has the following fields shown in Table 3-16.

Table 3-16 CARD.MMP Structure

Field Name	Field Description	Field Type	Size (Bytes)
itxlstx	Express terminal list headers	struct[128]	128*40
itxlsta	High priority terminal list headers	struct[128]	128*40
itxlstb	Low priority terminal list headers	struct[128]	128*40
iplimit	Port unack TX limit (in bytes)	uns.int [128]	128*4
ipbytes	Port unack TX size (in bytes)	int [128]	128*4
ipendg_bufs	Total pending Tx size (in bufrs)	int [128]	128*4
ipstat	Port status/Subchannel status	uns.short [128]	128*2
ipcntr	Terminal count per port	uns.short [128]	128*2
iportn	Port configuration bytes	struct[4]	4*4
isubch_buf	Subchannel configuration	struct[4]	4*256
ifcmdbuf	Current backplane command buffer	uns.char [16]	16
iscratch	Scratch area for buffer alignment	uns.char [64]	64
icfg_lkupt	Card configuration parameters	struct	64

CARD.BSP has the following fields shown in Table 3-17.

Table 3-17 CARD.BSP Structure

Field Name	Field Description	Field Type	Size (Bytes)
itxlstx	Express terminal list headers	struct[8]	8*40
itxlsta	High priority terminal list headers	struct[8]	8*40
itxlstb	Low priority terminal list headers	struct[8]	8*40
iplimit	Port unack TX limit (in bytes)	uns.int [8]	8*4
ipbytes	Port unack TX size (in bytes)	int [8]	8*4
ipstat	Port status/Subchannel status	uns.short [8]	8*2
ipcntr	Terminal count per port	uns.short [8]	8*2
iportn	Port configuration bytes	struct[8]	8*4
ifirqbp	$FIRQ reply buffer pointer	pointer	8
irespbp	Backplane response buffer pointer	pointer	8
istdtbp	$STDT reply buffer pointer	pointer	8
ifpntr	Pointer to current PTT	pointer	8
ifcmdbuf	Current backplane command buffer	uns.char [256]	256
ifirqbuf	FIRQ response buffer	uns.char [64]	64
irespbuf	Backplane response buffer	uns.char [64]	64
istdtbuf	STDT status buffer	uns.char [2048]	2048
iscratch	Scratch area for buffer alignment	uns.char [64]	64

The fields for **both** the *card.mmp* and *card.bsp* structures are described below. In many cases, the field names in the two structures are the same and in such cases, their function is identical. Only the number of elements in the array is different. The *itxlstx* through *iportn* fields are arrays indexed by the communication channel. For the 2/8-port ACC cards (e.g. *card.bsp*), each element in the array represents a physical

port. For the 4-port E1/T1 ACC cards, these arrays are indexed by a combination of port and subchannel. Specifically elements 0-31 represent port 0 and subchannels 0 to 31; elements 32-63 represent port 1 and subchannels 0 to 31, etc.

ITXLSTX - Express transmit list headers (*zptl_type*)

This field is an array of list headers for the express transmit requests associated with each port or subchannel of the Mux card. Each list header represents the head of a linked list of PTTs that have pending express transmit requests waiting to be sent to the Mux card. The list linkage is maintained through the PTTXLX pointer in each physical terminal table. The express transmit requests themselves are linked to the queue maintained by the PTTXQX field in each PTT. The structure of the list header is shown below. There is one copy of this list header for each port or subchannel on the Mux interface.

Table 3-18 ITXLSTX Express Transmit List Headers

Field Name	Field Description	Field Type	Size (Bytes)
nextl	Pointer to next active TX list	pointer	8
prevl	Pointer to previous active Tx list	pointer	8
tmcount	Number of terminals in the list	uns.short	2
subchn	Subchannel number of this list	uns.char	1
spare1	Reserved, unused	char	1
spare2	Reserved, unused	int	4
tmhead	Pointer to 1st terminal table in list	pointer	8
tmtail	Pointer to last terminal table in list	pointer	8

ITXLSTA - High priority transmit list headers (*zptl_type*)

This field is an array of list headers for the high priority transmit requests associated with each port or subchannel of the Mux card. Each list header represents the head of a linked list of PTTs that have pending high priority transmit requests waiting to be sent to the Mux card. The list headers in this array function identically to ITXLSTX only for the high priority transmit requests. A terminal may be linked to ITXLSTX, ITXLSTA and ITXLSTB all at the same time, if it has express, high, and low priority transmit requests in progress.

ITXLSTB - Low priority transmit list headers (*zptl_type*)

The list headers in this array function identically to ITXLSTA only for the low priority transmit requests. A terminal may be linked to ITXLSTX, ITXLSTA and ITXLSTB all at the same time, if it has express, high, and low priority transmit requests in progress.

IPLIMIT - Port unacknowledged transmit limit (in bytes)

This array contains the limit (in bytes) of all unacknowledged transmit messages to terminals belonging to the same port or subchannel of this interface. This array contains one element, one for each port or subchannel of the Mux card. If the DAM issues a transmit request to the Mux card and the Mux card does not have enough free buffers to hold the request data, the Mux will reject the request with an "*out of buffers*" error. The DAM will retry the request after an unacknowledged transmit request completes. This field is used to minimize the number of retries (and thereby increase overall performance) by imposing a limit on the size of unacknowledged transmit requests. This array is initialized by TTGEN from the default value *hpiplim* in the ZCOM header. This value is configured through the *Port-Limit or E1T1-Port-Limit* parameter in the TTGEN configuration file.

IPBYTES - Port unack TX size (in bytes)

This array is maintained by the DAM and keeps track of the number of 256-byte buffers currently in use by all unacknowledged transmit messages to a port of this interface card. The *ipbytes* array contains one element for each port/subchannel on the Mux card. The DAM compares the *iplimit* value for a port with the *ipbytes* value times 256 to determine whether it should issue a transmit request immediately or wait until a previously issued transmit request completes. The DAM uses this method to predict when the Mux card will reject a request with an "*out of buffers*" error and can thereby avoid sending a request now which has a high probability of needing to be retransmitted later.

IPENDG_BUFS - Total pending transmit size (in # of buffers)

This field is used internally by the ZCOM drivers.

IPSTAT - Port/Subchannel status

Depending on the card type this interface table is for, this ipstat field either contains the status of each **subchannel** in a 4-port E1/T1 ACC card, or the status of each **port** in a non-4-port E1/T1 ACC card (e.g., 2-port card or 8-port ACC card).

The individual ipstat bits are handled as follows:

ZCOM_IPSTAT_UND:
The undefined bit. Being undefined is synonymous with being not configured.
For a non-4-port card, this bit in ipstat indicates whether a port is defined or not.
For a 4-port card, this bit in ipstat indicates whether a subchannel is defined or not. However, for defined/undefined status, we must also keep track of whether ports are defined or not. So we will use another field, iportn, to indicate whether the ports these subchannels are related to are defined or not. If iportn[] is equal to -1, then the port is undefined. Note in the case of a 4-port card that if a port is marked as undefined, then all subchannels related to that port are marked as undefined. If a port is marked as defined, a subchannel may still be marked undefined until it is explicitly marked as defined. So for a 4-port card we need a way to keep separate track of whether ports and subchannels are defined or not.

ZCOM_IPSTAT_DWN:
The down bit. Indicates the status of ports.
For a non-4-port card, this bit in ipstat indicates whether a port is down or not.
For a 4-port card, the down status for the subchannels related to a port as a group are being used to indicate the down status of the port. For example, if a port is considered up, then all subchannels related to that port are marked as up in ipstat. If a port is considered down, then all subchannels related to that port are marked as down in ipstat. Individual subchannels related to that port cannot be marked up or down independent of the other subchannels for that port.

ZCOM_IPSTAT_DSB:
The disabled bit.
For a non-4-port card, this bit in ipstat indicates whether a port is disabled or not.
For a 4-port card, this bit in ipstat indicates whether a subchannel is disabled or not. Individual subchannels can be marked as disabled or enabled, independent of each other. For a 4-port card, a request may be sent down to the ZCOM software to enable or disable a “port”. This just means marking all of the subchannels for that port as enabled or disabled. However, for 4-port cards, the ZCOM software does not keep track of the enabled/disabled status for ports. It only keeps track of the enabled/disabled status for subchannels via ipstat.

ZCOM_IPSTAT_TXB:
The out-of-transmit buffers bit.
For a non-4-port card, indicates the status of ports.
For a 4-port card, indicates the status of subchannels.

ZCOM_IPSTAT_TXBX:

The out-of-transmit-buffers (XPS queue) bit.
For a non-4-port card, indicates the status of ports.
For a 4-port card, indicates the status of subchannels.
The DAM maintains the port status to determine the usability of a port/subchannel. A port/subchannel is operating normally if the status value is zero. If it is out of transmit buffers, no new transmit data transfer will be initiated. If the port/subchannel is disabled, down or undefined, the DAM will ignore all processing for this port.

IPCNTR - Terminal count per port/subchannel

This field contains the number of terminals defined on each port/subchannel of this Mux interface card. It is initialized by TTGEN based on the number of *Term* entries in the TTGEN configuration file and is kept for informational purposes only.

IPORTN - Port configuration buffer (*zpconf_type*)

This array contains the configuration bytes for each port on the Mux card. The *Port* statements in the *Port-Definition* section of the TTGEN configuration file are used to define the initial values of this array. Each element of the array is associated with a single port on the Mux card. Each element can be dynamically configured by calling the *zport* routine. The layout is similar to the data portion of the $PORT backplane transaction command for the 2/8-channel ACC cards and is as shown in Table 3-19.

Table 3-19 Port Configuration Buffer IPORTN

31-30	29-28	27-26	25-24	23-22	21-20	19-16	15-14	13-0
ecode	sync	mode	parity	xclock	sclock	baud	pmode	spare

The layout for the 4-channel E1/T1 ACC card is as follows:

Table 3-20 4-Channel E1/T1 ACC Card Format

31-30	29-26	26-22	21-20	19-16	15-11	10-0
ecode	fmode	spare	sclock	spare	pmode	spare

ecode - Encoding modes

sync - Sync mode select

mode - Operating mode

parity - Parity select

xclock - Clock multiplier

sclock - Clock source

baud - Baud rate value

pmode - Port mode select

fmode - Frame mode select

spare - Reserve, not used

ISUBCH_BUF - Port subchannel configuration buffer (*subchbuf_def*)

This array contains the subchannel configuration bytes for each port on the ACC E1/T1 4-port Mux card. The *Subch* statements in the *Subchannel-Definition* section of the TTGEN configuration file are used to define the initial values of this array. Each element of the array is associated with a single port on the Mux card. Each element can be dynamically configured by calling the *zconfig* routine. The structure layout is as shown in Table 3-21.

Table 3-21 Port Subchannel Configuration Structure

Field Name	Field Description	Field Type	Size (Bytes)
tmsl	Timeslot assignment array	uns.int[32]	32*4
spec	Subchannel specification array	uns.int[32]	32*4

Where:

Each element of the timeslot array has the structure shown in Table 3-22.

Table 3-22 Timeslot Array Element Structure

31-30	29	28-24	23-16	15-14	13	12-8	7-0
spare	TTI	TxSubch	TxFill	spare	RTI	RxSubch	RxFill

TTI - Transmit Timeslot Inhibit

TxSubch - Transmit subchannel number

TxFill - Transmit fill mask

RTI - Receive Timeslot Inhibit

xclock - Clock multiplier

RxSubch - Receive subchannel number

RxFill - Receive fill mask

spare - Reserved, not used (must be 0)

Each element of the subchannel specification array has the following structure:

Table 3-23 Subchannel Specification Array Element Structure

31-30	29-24	23-8	7	6-3	2-1	0
spare	ITBS	spare	INV	spare	mode	spare

ITBS - Individual Transmit Buffer Size

INV - Inversion flag

mode - Transmission mode

spare - Reserved, not used (must be 0)

IFIRQBP, IRESPBP, ISTDTBP - DMA reply buffer pointers (char *)

These fields point to the corresponding buffers kept at the end of the interface table that are used by the DAM for DMA operations. The actual values of these pointers are initialized at ZCOM subsystem initialization time by the LDM. These pointers are chosen by the LDM so that each buffer is 64-byte aligned. (The buffers must be aligned so that the DMA may read/write directly to them). These fields are not used for ACC E1/T1 cards.

IFPNTR - Pointer to active physical terminal table

This field contains the address of the currently active physical terminal table entry within this interface table. This field is not used for ACC E1/T1 cards It is maintained by the DAM.

IFCMDBUF - Current backplane command buffer

This field is the buffer area for DMA transfer of a backplane command to the Mux interface card. The actual area being used is offset by the number of bytes specified in IFBOFF. This area is accessed by the pointer field IFCMDBP.

IFIRQBUF, IRESPBUF, ISTDTBUF - DMA response buffers

These are response buffers that are written to by DMA read requests to the Mux interface card. The FIRQ buffer contains the results of an $FIRQ backplane transaction. The Response buffer contains the response that provides the DAM with the completion status from the

Mux card firmware. The Status buffer contains the data returned from a $STDT transaction. This data is updated into the physical terminal tables by the DAM. The actual area being used is offset by the number of bytes specified in IFBOFF. This area is accessed by the pointer set up in the corresponding pointer fields IFIRQBP, IRESPBP, ISTDTBP. These fields are not used for ACC E1/T1 cards.

ISCRATCH - Scratch area for buffer alignment

This area is needed to insure that DMA buffers can be aligned on a 64-byte boundary. This is a requirement of the hardware.

IFPLOOK - Physical terminal table lookup

This table is used by the DAM during FIRQ processing to determine the physical terminal table address based on the port and terminal number passed back from the Mux interface. It is indexed by the value [(port number * 8) + terminal number] for the 2/8-channel NIO and EISA ACC cards. For the ACC E1/T1 and 8-Port PCI cards, it is simply indexed by the terminal number. The DAM uses this table as an interrupt vector to the PTT associated with an unsolicited interrupt (FIRQ). This table is variable in size and is dynamically allocated from HP-UX kernel memory. Usually only a few of the values in the array are used. Unused entries are null filled. The LDM creates and initializes this table during startup of the ZCOM subsystem. This table is not accessible by user space applications.

This field is an array of pointers, which will be accessed by an array index.

ICFG_LKUPT - Optional E1/T1 tunable parameters (zcfg_lookup_t)

This structure contains the optional E1/T1 tunable parameters that allow you to fine tune the allocation of resources (memory) within the E1/T1 card. Specifically, these parameters allow you to fine tune how card memory is divided up between internal tables and I/O buffers. These parameters may be specified in the TTGEN *Interface-Definition* section. This structure has the fields shown in Table 3-24.

Table 3-24 Optional E1/T1 Tunable Parameters

Field Name	Field Description	Field Type	Size (Bytes)
datadelay	$DATA transfer delay (milliseconds)	uns.short	2
ttablenbr	Maximum # of terminal table entries	uns.int	2
ttablesize	Size of terminal table entries (in bytes)	uns.int	4
zbufsize	Size of the card's buffers (in bytes)	uns.int	4
zbufnbr	Number of I/O buffers	uns.int	4
mheadnbr	Number of buffer header entries	uns.int	4
mquota	Quota of msg headers allowed for each subchannel	uns.int	4

Table 3-24 Optional E1/T1 Tunable Parameters (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
bquota	Quota of buffers allowed for each subchannel	uns.int	4
tracesize	Firmware trace buffer size	uns.int	4
traceopt	Firmware trace option bits	uns.int	4
spare1~3	Reserved, not used	uns.int[3]	4*3
tsize	Size of timer tables in bytes	uns.int[4]	4*4

Response Records

Response records are used to keep track of the issue-and-wait operations in the ZCOM subsystem. Currently, it is used for the send-with-wait (for example, *zsend mode 8*), port configuration (*zport* and *zconfig* routines), interface card control (*zconfig*), and remote API calls.

When a program requests a issue-and-wait operation, the LDM allocates a spare response record from the pool, sets up the sleeping address, and puts the caller to sleep. At a later time, when the operation is completed, the driver (LDM or DAM) sets up the return status and wakes up the waiter. The waiter then reads the return status and releases the response record back to the pool.

The layout of the response record (*zrsp_type*) is shown below.

Table 3-25 Response Record Format

Field Name	Field Description	Field Type	Size (Bytes)
RPADDR	Waiter sleeping address	pointer	8
RPTYPE	Expected response type	uns.short	2
RPNODE	Remote node	uns.short	2
RPSEQN	Request/Response sequence number	uns.int	4
RPEXPT	Response expiration time	uns.int	4
RPCODE	Return status/error code	int	4
RPBUFQ	Response buffer queue	struct	56

RPADDR - Waiter sleeping address

The LDM puts an appropriate address into this field and puts the application to sleep on this address. This field is zero when the record is not being used. When this field is set, the remaining fields of the structure will contain valid information.

RPTYPE - Expected response type

This field contains the type of response the waiter is expecting. The LDM sets up the response type when it initializes the record. This field is validated when a response is returned. Currently, it may contain one of the values in Table 3-26.

Table 3-26 Response Type Field

Symbol	Value	Meaning
ZCOM_RPTYPE_BUFFER	1	Buffer completion response
ZCOM_RPTYPE_REMOTE	2	Remote completion response
ZCOM_RPTYPE_IRR	3	Interface request response

A buffer completion response is returned when a local request (e.g. port configuration) is completed. A remote completion response is returned when a remote API request is processed in the remote system. An interface request response is returned when ZCOM has completed an interface control request.

RPNODE - Remote node

This contains the remote node number for a remote completion response. It is initialized by the LDM before a remote request is sent, and is used to validate the returned response.

RPSEQN - Request/Response sequence number

For remote and interface requests, the LDM generates and stores a request sequence number in this field. This number is used to validate the returned response.

RPEXPT - Response expiration timeout value

This filed contains the expiration time (Coordinated Universal Time, in seconds) of an outstanding request. The LDM generates a timeout response if the current time exceeds this value. Currently, only the remote completion response uses this field to trigger timeout processing.

RPCODE - Return status or error code

When a response is received, the response status or error code is temporarily stored in this field. The value is then picked up by the waiting process when it resumes execution.

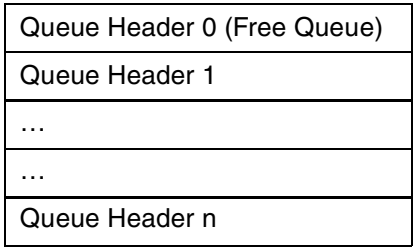
RPBUFQ - Response buffer queue

When a response carries data in addition to just a status or error code, the data is stored in this queue temporarily. The data is retrieved by the waiting process when it resumes execution. Currently, only remote completion responses use this queue for data storage.

Queue Header

The queue header area contains the descriptor block for all the ZCOM subsystem queues. The first queue is used to link the free buffers in the buffer pool.

Figure 3-7 Queue Header Area



Free Queue Header (zfqh_type), queue 0, basically is the same as the other buffer queues. However, its fields are viewed and used slightly differently from the other queues. Its size is the same as a buffer queue.

Table 3-27 Free Queue Header Structure (zfqh_type)

Field Name	Field Description	Field Type	Size (Bytes)
QNMSG	Number of messages on queue	uns.int	4
QALLOC	Number of allocated messages	uns.int	4
QHEAD	Pointer to first message on queue	pointer	8
QTAIL	Pointer to last message on queue	pointer	8
QMMAX	Historical max number of messages	uns.int	4
QTMSG	Total messages through queue	uns.int	4
QBYTES	Number of bytes on queue	uns.int	4
QWAITER	Number of waiters on this queue	uns.short	2
QFLAG	Queue flags	uns.short	2

Table 3-28 Queue Header Structure (zqhd_type)

Field Name	Field Description	Field Type	Size (Bytes)
QNMSG	Number of messages on queue	uns.int	4
QLIMIT	Max messages/bytes allowed on queue	uns.int	4
QHEAD	Pointer to first message on queue	pointer	8
QTAIL	Pointer to last message on queue	pointer	8
QMMAX	Historical max number of messages	uns.int	4
QTMSG	Total messages through queue	uns.int	4
QBYTES	Number of bytes on queue	uns.int	4
QWAITER	Number of waiters on this queue	uns.short	2
QFLAG	Queue flags	uns.short	2
QFDATA	Queue function data	uns.int	4
SPARE1	Reserved, not used	int	4
QFUNC	Queue function	pointer	8

QNMSG - Number on queue

Counter of the number of messages on the queue. When this is zero the queue is empty.

QALLOC - Number of allocated messages

This field is used in the free queue header to keep track of the number of buffer segments allocated for ZCOM subsystem use. A value of zero means that no buffers have been allocated for use by the ZCOM subsystem. A large value indicates heavy usage of the buffer pool.

QLIMIT - Queue limit

The maximum number of messages or bytes that may be linked to this queue. If the ZCOM_QFLAG_BLT bit in *qflag* is not set, this field is a limit on the number of messages. Otherwise, this value is a limit on the number of bytes that can be linked to this queue. The default queue limit

is initialized from the *Queue-Limit* parameter in the TTGEN configuration file when the queue is initialized. The queue limit may be modified dynamically by calling the *zsetql* routine.

QHEAD, QTAIL - Queue head and tail pointers (zbhd_type *)

These are pointers to the first and last messages attached to this queue. If the queue is empty, these values are undefined. If the queue has only one buffer, then the values are equal.

QMMAX - Number of messages on the queue

The largest number of outstanding messages ever linked to this queue at any given point in time. This value is initialized to zero and is updated any time QNMSG is larger than this field.

QTMSG - Total number of messages through this queue

Counter of all messages that have either passed through, or are still on, this queue.

QBYTES - Number of bytes on the queue

The number of bytes on the queue is the actual number of bytes of all the buffers (messages) linked to this queue. This includes the data length and buffer header areas of each message in the queue. The sum of these fields across all queues (including the free queue) at any instant will equal the total amount of buffer space allocated in the ZCOM subsystem.

QWAITER - Number of processes waiting on this queue

This field is used to indicate whether any processes are waiting for data from this queue. If a process reads from the queue and no data is available, and it has not done a read without wait, the process is suspended (put to sleep) by the LDM and this counter is incremented. At a later time when a new message is added to this queue, the LDM or DAM will wake up all processes suspended on this queue address and set this field to zero.

QFLAG - Queue flags

This field contains some indicators which may affect the processing of a queue. The format is shown in Table 3-29.

Table 3-29 Queue Flag Format

15-8	7	6	5	4	3	2	1	0
				COL	SEL	FUNC	BLT	ENB

Symbol	Value (hex)	Meaning
ZCOM_QFLAG_ENB	0x0001	Queue is enabled
ZCOM_QFLAG_BLT	0x0002	Queue limit on byte size
ZCOM_QFLAG_FUNC	0x0004	Queue uses wakeup function
ZCOM_QFLAG_SEL	0x0008	Queue has a failed “select”
ZCOM_QFLAG_COL	0x0010	Queue has a “select” collision

When the FUNC bit is set, it indicates the kernel function kept in *qfunc* is to be called whenever data is added to the queue. This feature may only be used by kernel drivers (via the *zsetkfunc* routine).

The BLT bit is set if the queue limit check is performed on the number of bytes rather than the number of messages. Currently, only the PTT unacknowledged transmit queue uses it.

Queue manipulation is allowed only when the ENB bit is set. All queues are set to disabled by TTGEN initially, and enabled by the *zopen( )* call (where a buffer queue is assigned to a ZLU). During ZCOM subsystem shutdown, the LDM will disable the program queues by clearing this bit.

The SEL and COL bits are used internally for the “select-for-read” function on program ZLUs. When a program ZLU has been “selected” while it has no data, the LDM sets the SEL bit. If more than one program selects this empty ZLU, the COL bit is set. When data arrives for this ZLU the LDM uses these two bits to decide whether to wake up one or multiple programs waiting on the “select” for this ZLU.

QFDATA - Queue function data

QFUNC - Queue function

These two fields have valid data only when the ZCOM_QFLAG_FUNC bit in *qflag* is set. They are set up by the *setkfunc* routine and contain a pointer to a kernel function and a 32-bit value for use in calling the function. Each is only used by the kernel so that a kernel function is called whenever data is added to the queue.

When the ZCOM_QFLAG_FUNC bit is not set, the QFUNC field may be used by the LDM as temporary storage for the process pointer to support the program ZLU *select-for-read* function.

Data Buffer Pool

The data buffer pool is part of the ZCOM memory structure. The size of data buffer pool is defined in the TTGEN configuration file. The data buffer pool is a global resource within the ZCOM subsystem.

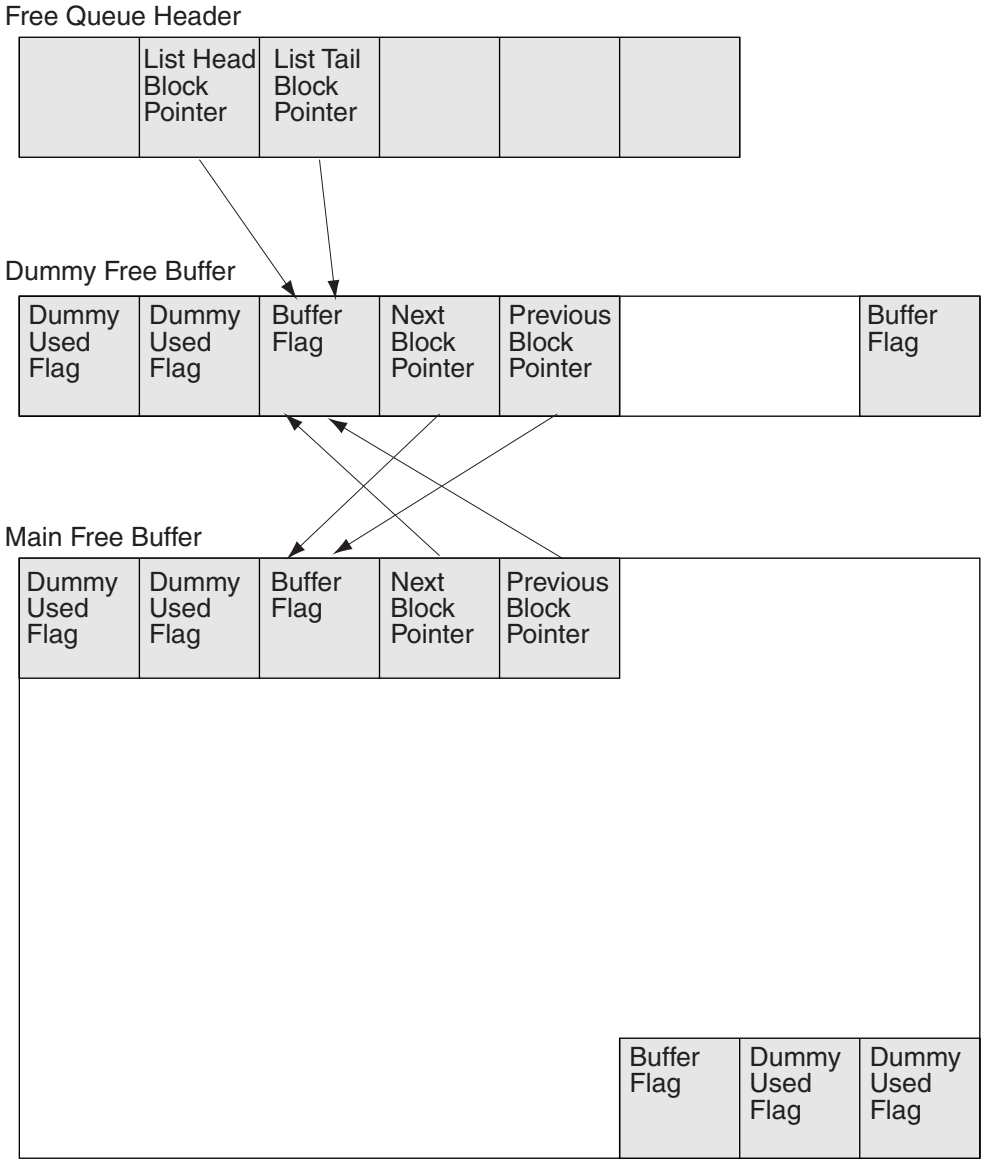
Organization of Buffer Pool

The buffer pool is a contiguous block of memory divided dynamically into a number of variable length buffers. When the ZCOM subsystem is initialized, the single buffer pool is divided into 2 buffers: one small dummy buffer which stays in the free queue all the time, and a main buffer that contain all the remaining free space from the memory block. The size of the dummy buffer is smaller than the minimum size in buffer allocation, so it will never be released from the free pool. The two buffers are linked together in a circular list. This arrangement is to minimize the updates to the free queue header (see next paragraph), hence the buffer management code can be more efficient.

The access to the buffer pool memory block is via the free queue header. The free queue header is the first queue header (header 0). The header contains a pointer to the first free block, the last free block, and some statistics that relate to the use of the free pool. These include the number

of blocks in the pool (initially 2), the total number of bytes in the pool, and the number of accesses to the pool to get a memory buffer. The buffer pool structure is shown in Figure 3-8.

Figure 3-8 Buffer Pool Layout (when first initialized)



Free Buffer Header Structure *(zfbh_type)*

The buffer header contains information about the buffer as well as a linkage to the next buffer. *(zbhd_type)*. See Table 3-30 and Table 3-31.

Table 3-30 Free Buffer Header Structure *(zfbh_type)*

Field Name	Field Description	Field Type	Size (Bytes)
BFFLAG	System flags (length & status)	uns.int	4
SPARE1	Reserved, not used	int	4
QHEAD	Pointer to next buffer on queue	pointer	8
BFPREV	Pointer to previous buffer on queue	pointer	8

Table 3-31 Buffer Structure *(zbhd_type)*

Field Name	Field Description	Field Type	Size (Bytes)
BFFLAG	System flags (length & status)	uns.int	4
BFLEN	Buffer length in bytes (data portion)	int	4
BFLINK	Pointer to next buffer on queue	pointer	8
BFDATA	Pointer to 1st byte of data in buffer	pointer	8
BFRESP	Pointer to response record	pointer	8

BFFLAG - Buffer flags

The use of the memory pool is based around a *buffer flag* field that indicates the size of a memory block as well as containing a flag that indicates whether the block is currently in use. This field appears at the start and end of each buffer segment. This field is 32 bits long. The size of the buffer is always a multiple of 8-bytes (so each buffer is 64-bit aligned). This allows the low order bit to be used as the *Used flag* (ZCOM-BFFLAG-BUSY). This field is located at both the start and end of the block, so that adjacent blocks can be identified from either end of any block.

Table 3-32 Buffer flags BFFLAG Field Format

31	1	0
Block Size		
		Used flag

Block size The size of the memory block in bytes (including buffer flags). The block size is always a multiple of 8 bytes. However, the lower (least significant) bit is used to indicate whether the block is used or not, and is not included in the size of the block.

Used flag =0 Block is unused,

=1 Block is used

Dummy Used flag - Start and end indicator of memory pool

This is a dummy *buffer flag* at the beginning and end of the buffer pool area. The used flag is set to indicate this block is used, so the coalesce algorithm will think there is a used block before the start of the pool, and after the end of the pool. The size is set to 0xFFFFFFFE so that these dummy blocks can be distinguished from the legitimate ones.

BFNEXT - Pointer to next buffer on queue

BFPREV - Pointer to previous buffer on queue

These pointers are used to link all the free blocks within the memory pool, to form a circular list. The free queue contains the initial pointer to the first and last free block. The value in the pointer is the address of the *buffer flag* (i.e., the address of the *next block pointer* in the block).

BFLEN - Buffer length in bytes (data portion)

This field contains the usable memory size (in bytes) within the buffer. The usable area of a buffer is indicated by its location, BFDATA, and its size, BFLEN.

BFLINK - Pointer to the next buffer on queue

This is the buffer linkage to the next buffer in the chain. It contains the address of the first field (i.e., BFFLAG) of the next buffer. If this is the last buffer on the list, it contains NULL.

BFDATA - Pointer to the buffer data area

This field contains the memory address where the message header resides within the current buffer. This memory address is set up such that the message data, which follows the message header, will be aligned properly depending on its uses (e.g. for inbound DMA, it must be on a 64-byte boundary). Usually, a buffer will have unused area before and after the message. In case of buffer for inbound DMA operation, the sufficient unused area is allocated to satisfy the DMA operation without overflow into the next buffer.

This field is set up by the ZCOM subsystem when a buffer is allocated from the buffer pool and will not be modified until the buffer is released to the buffer pool.

BFRESP - Pointer to response record

This field is valid only when the buffer is associated with a send-and-wait operation. It contains the address of the response record for the waiting program. On completion of such a request, the LDM uses this field to locate the response record, where the return status is placed.

For other types of operation, this field contains invalid data.

Message Header

In the buffer header (i.e., BFDATA pointer), there is a message header that describes the message type, the source and destination addresses, and the type of response expected from the message. The message header is the first 30 bytes of the buffer data area. When the data buffer is allocated, the initial data offset is chosen so that the message data starts on a specific boundary depending on its purpose (e.g. for a inbound DMA buffer, the data area must be on a 64-byte boundary). For inbound DMA buffer, enough space is left at the end of the buffer so that the final 64-byte boundary is within the buffer area but beyond the end of the data area. These boundary limits ensure that the DMA reads and writes will not cause side effects by corrupting other buffers. Besides, the size of the buffer is rounded up to a multiple of 8-bytes, to satisfy the 64-bit alignment which is needed for the 64-bit kernel.

Message Header (*zmhd_type*)

Table 3-33 shows the information in the message header (*zmhd_type*).

Table 3-33 Message Header (*zmhd_type*)

Field Name	Field Description	Field Type	Size (Bytes)
MID	Message ID header	struct	24
MRQ	Message request block	struct	6

Message ID Header (*zmid_type*)

The Message ID Header contains information used for delivery of the message. See Table 3-34.

Table 3-34 Message ID Header (*zmid_type*)

Field Name	Field Description	Field Type	Size (Bytes)
MSTYPE	Message type	uns.char	1
MSRESP	Message response code	uns.char	1
MZDEST	Destination ZCOM address	struct	6
MZSRCE	Source ZCOM address	struct	6
MTAGW1	Message tag word 1	uns.short	2
MTAGW2	Message tag word 2	uns.short	6
MZAUXL	Auxiliary ZCOM address	struct	6

MSTYPE - Message type

This field contains the message type. If bit 3 of the message type is set (ZCOM_MSTYPE_REMOTE (0x08)), then the message is remote. Notice that the remote message type maps onto the message type for a local terminal.

Table 3-35 Message Type MSTYPE

ZCOM_MSTYPE_REMOTE	0X08	Mask for remote message bit
	0	Undefined, reason in IERR
ZCOM_MSTYPE_MSLT	1	Message from local terminal
ZCOM_MSTYPE_LPLP	2	Msg from local program to program
ZCOM_MSTYPE_TDLZ	3	Timeout from local ZLU
ZCOM_MSTYPE_LPLT	4	Msg from local program to a terminal
ZCOM_MSTYPE_LSEM	5	Local system event message
ZCOM_MSTYPE_RSLT	6	Response message from local terminal
ZCOM_MSTYPE_RSLP	7	Response message from local program
	8	Not defined
ZCOM_MSTYPE_MSRT	9	Message from remote terminal
ZCOM_MSTYPE_RPLP	10	Message from remote program to a program
ZCOM_MSTYPE_TORZ	11	Timeout from remote ZLU
ZCOM_MSTYPE_RPLT	12	Message from remote program to a terminal
ZCOM_MSTYPE_RSEM	13	Remote system event message
ZCOM_MSTYPE_RSRT	14	Response msg from remote terminal
ZCOM_MSTYPE_RSRP	15	Response msg from remote program

MSRESP- Response code

The response code is used in two ways. Firstly, for message types 2, 4, 10, and 12 (send message), it indicates the type of response expected when this message is sent or delivered. Secondly, for other message types, it is the status of the completed message.

Response code for message type 2 and 10 are shown in Table 3-36.

Table 3-36 Response code for message type 2 and 10 Format

7	6	5	4	3	2	1	0
LPR	XPS			PGW	BFR	DEF	ERR

ERR	Response if error exists
DEF	Definite response (error or no error)
BFR	Return status and message buffer as response (for data message only)
PGW	Program is waiting for response
XPS	Express transmit message
LPR	Low priority transmit message

Response code for all other message types are shown in Table 3-37.

Table 3-37 Response code for all other message types

7	6	5	4	3	2	1	0
				ZCOM error code or 0			

MZDEST - Destination ZCOM address

This is the ZCOM address to which the message is to be sent.

MZSRCE - Source ZCOM address

This is the ZCOM address of the sender of the message. This field is returned in the message header with the *zread* routine. When a message is received from a terminal, the sender *zaddr.zlu* is the ZLU number of the terminal.

MTAGW1, MTAGW2 - Message tag words

These are application defined tag words that are passed unaltered through the ZCOM subsystem with the message data. They are supplied in the *zsend* and *zcntl* calls and returned in the *zread* call. When a receive message is returned from a terminal, the tag words are set to zero.

MZAUXL - Auxiliary ZCOM address

When using the multiplexed mode of operation through the logical terminal table, this field stores the original source ZLU.

Message Request Header (*zmrq_type*)

The six-byte message request block may or may not be present, depending on the message type and whether it is destined for a terminal. For request types 1, 4, and 6, to a terminal, the request block will be present; for request types 2 and 7 the request block will contain meaningless data. More information on the contents of this block can be found in the firmware reference manual. The Message Request Header is always present. For message types 2, 3, 7, 10, 11, and 15, it contains meaningless data. For types 4, 6, 12, and 14, it contains valid data.

Table 3-38 Message Request Header (*zmrq_type*)

Field Name	Field Description	Field Type	Size (Bytes)
MRQCODE	Request code	uns.char	1
MRQTERM	Terminal number	uns.char	1
MRQSTAT	Request status	uns.char	1
MRQTAG	Tag parameter	uns.char	1
MRQLEN	Data length	uns.short	2

MRQCODE - Terminal request code

This field contains the request code sent to or received from the Mux firmware on the interface card. The message request code values are shown in Table 3-39.

Table 3-39 MRQCODE Terminal request codes

Request	Value	Description
ZCOM_MRQCODE_UND	0	Undefined type
ZCOM_MRQCODE_READ	1	Terminal input
ZCOM_MRQCODE_WRITE	2	Terminal write
ZCOM_MRQCODE_CNTWR	3	Terminal write
ZCOM_MRQCODE_STDT	4	(Internal Use) status transfer
ZCOM_MRQCODE_STATUS	5	Unsolicited status change
ZCOM_MRQCODE_TERM	6	Set terminal parameters
ZCOM_MRQCODE_ENB	7	Terminal enable
ZCOM_MRQCODE_DSB	8	Terminal disable
ZCOM_MRQCODE_ACT	9	Message from remote terminal
ZCOM_MRQCODE_DEA	10	Terminal deactivate
ZCOM_MRQCODE_DEL	11	Terminal delete
ZCOM_MRQCODE_PORT	14	Port configuration
ZCOM_MRQCODE_MULTI	15	Multi-completion RXDT request code

MRQTERM - Terminal number

This is the relative terminal number within the Mux card. It comes from the PTTERM field in PTT.

MRQSTAT - Request status

This is the status code returned from the protocol module and may be protocol type dependent.

MRQTAG - Tag parameter

This tag parameter is used to carry protocol specific information from the application program to the protocol module on the Mux card and return information from the firmware to the driver or application program.

MRQLEN - Data length

This is the total length of the data to be sent, or the length of data received, in this message.

System Event Message (zevent_type)

A System Event Message (SEM) is a special message generated by the ZCOM subsystem to represent an alarm condition or type of event that has occurred within the ZCOM subsystem. Applications may request that ZCOM deliver these events to a program ZLU by issuing a *zevent_rcvr* call. A SEM is returned to the application as a normal data message with a message type of ZCOM_MSTYPE_LSEM (5) or ZCOM_MSTYPE_RSEM (13). See the *zevent_rcvr* routine for addition details.

The structure used to hold the system event messages is composed of a fixed header structure and a variable part that is specific to the type of event. See Table 3-40.

Table 3-40 System Event Message (zevent_type) Structure

Field Name	Field Description	Field Type	Size (Bytes)
SETYPE	System Event Type	uns.char	1
SEFLAG	Event flags (currently not used)	uns.char	1
SENODE	Node where event was generated	uns.short	2
SETIME	Time when the event occurred	uns.int	4
SEDATA	Event specific data	union struct	8

SETYPE - System Event Type

This field contains the type of event represented by this message. Currently there are three defined types of events; a node status event (ZCOM_SETYPE_NODE), a card status change event (ZCOM_SETYPE_CARD), and a dynamic reconfiguration event (ZCOM_SETYPE_RECONFIG). The format of the variable part of the message is dependent on the event type.

SEFLAG - Event flags (currently not used)

This field is currently not used. It is intended to contain global information on the event (e.g. event priority, etc).

SENODE - Node where event was generated

This field will contain the ZCOM node number of the system that generated the event message. If the message type is ZCOM_MSTYPE_LSEM, this will contain the local ZCOM node number.

SETIME - Time when the event occurred

This field contains the Coordinated Universal Time (UTC) of when the event occurred. It is taken from the originating system at the time of the event.

SEDATA - Event specific data

Immediately following the header is a union that contains information specific to the type of event that occurred. There are currently two different event types that can be generated. The data structure for each is documented below.

ZCOM_SETYPE_NODE - Node status change event

The node status change event is generated whenever a node changes state or the state of the ZNODE daemon changes. The event data is referenced using *sedata.node.<field name>* and has the following fields:

Table 3-41 Node Status Change Event Fields (ZCOM_SETYPE_NODE)

Field Name	Field Description	Field Type	Size (Bytes)
NODE_NUM	Node Number	uns.short	2
NSTAT	Node status change indicator	char	1
SPARE	Reserved, not used	char	1
HSTAT	Host link status change indicator	char [4]	4

NODE_NUM - Node number

This is the number of the node that has changed state. Note that this node is different from the SENODE field. For example, in ZCOM node 10, two remote nodes 40 and 50 are defined. In node 10, if the system detects that remote node 40 has gone down, the SEM generated will have SENODE=10, SEDATA.NODE.NODE_NUM=40.

NSTAT - Node status change indicator

This field contains a numeric value indicating the status change. If the value is zero, then there has been no change to the node's status. If the value is +1, the node changed state from DOWN to UP. If the value is -1, the node changed state from UP to DOWN.

HSTAT - Host link status change indicator

This field is an array of four values, one for each potential link associated with NODE_NUM. Each entry indicates the status change of one link. If the value is zero, then there has been no change to the link's status. If the value is +1, the link changed state from DOWN to UP. If the value is -1, the link changed state from UP to DOWN.

ZCOM_SETYPE_RECONFIG - Dynamic Reconfiguration Event

The dynamic reconfiguration event is generated by using the *zconfig* routine, typically after programmatically reconfiguring the ZCOM subsystem during runtime operations. The event data is referenced using *sedata.recfg.<field name>* and has the fields shown in Table 3-42.

Table 3-42 Dynamic Reconfiguration Event (ZCOM_SETYPE_RECONFIG)

Field Name	Field Description	Field Type	Size (Bytes)
CFG_CLASS	Type of reconfiguration performed	uns.int	4
CFG_INFO1	Caller supplied data	uns.int	4

CFG_CLASS - Type of reconfiguration performed

The field should contain the type of reconfiguration performed. Values from 0 to 999 are reserved for use by the ZCOM subsystem. There is currently one type defined, ZcCLASS_X25 (1), used to indicate that an X.25 link has been reconfigured.

CFG_INFO1 - Caller supplied data

The application or driver calling *zconfig* to generate the dynamic reconfiguration SEM should supply a class specific piece of data. In the case where the CFG_CLASS is set to ZcCLASS_X25, this field will contain the X.25 level 2 ZLU that was reconfigured.

ZCOM_SETYPE_CARD - Change in Card Status Event

This event is generated whenever there is a change in the state of an ACC interface card. This includes notification of firmware failures (causes an automatic restart of the card), user initiated restarts, hardware failures, and for those systems that support it, the on-line addition, removal, or replacement of an ACC card. The event data is referenced using *sedata.card.<field name>* and has the following fields:

Table 3-43 Change in Card Status Event (ZCOM_SETYPE_CARD)

Field Name	Field Description	Field Type	Size (Bytes)
reason	Reason for the change in card state	uns.char	1
iftno	ttgen mux# (if any)	uns.int	1
bus_addr_len	Number of entries in bus_addr	uns.int	1
bus_addr[7]	Bus address path to card	uns.short	7 *2
card_addr	Card's slot address	uns.short	2

Reason - The reason for the change in card state may be one of the following:

ZcSE_CARD_ADDED - An ACC card has been added to the system at the H/W path indicated by the bus_addr[] and card_addr fields

ZcSE_CARD_SUSPENDED - The operator has suspended card operation in preparation for on-line replacement of the card. The iftno field indicates which ACC card was suspended.

ZcSE_CARD_REMOVED - The ACC card indicated by the iftno field has been removed from the system and is no longer usable.

ZcSE_CARD_REPLACED - The ACC card indicated by the iftno field has been replaced (on-line) and can now be used.

ZcSE_CARD_STOPPED - The ACC card indicated by the iftno field has been halt by user request and must not be used.

ZcSE_CARD_FAILED - The ACC card indicated by the iftno field has been failed and is no longer usable.

ZcSE_CARD_USER_RESTART - The ACC card indicated by the iftno field is being restarted by user request and must not be used until a ZcSE_CARD_RESTRT_CMPLT event is received.

ZcSE_CARD_FAIL_RESTART - the ACC card indicated by the iftno field is being restarted due to a firmware failure and must not be used until a ZcSE_CARD_RESTRT_CMPLT event is received.

ZcSE_CARD_RESTRT_CMPLT - The ACC card indicated by the iftno field has been redownloaded and restarted. It can now be used by applications.

Iftno - This is the ZCOM MUX number of the card that has changed state.

Bus_addr_len - Number of bus addresses in the bus_addr[] field. This field is only valid when the event has a reason code of ZcSE_CARD_ADDED.

Bus_addr[] - Hardware path of the ACC card being added. This field is only valid when the event has a reason code of ZcSE_CARD_ADDED.

Card_addr - Contains the address of the slot that the card was inserted into. This field is only valid when the event has a reason code of ZcSE_CARD_ADDED.

ZCOM Kernel Data

This is a kernel data structure shared by the LDM and DAM. Unlike the ZCOM subsystem memory tables (which are set up when ZCOM subsystem is started), this data structure is generated into the HP-UX operating system with the LDM and always exists even when the ZCOM subsystem is not started up. The kernel entry point for this data structure is *Zcom* which consists of four components, NCARD, HPATBL, SYS, and HP. These components are described below.

NCARD

NCARD (Number of Mux interface cards installed in the system) is a counter used to indicate the number of Mux interface cards in the host computer. It is updated by the DAM when the Mux interface cards are bound to the DAM during system boot-up.

HPA Table

The Hard Physical Address (HPA) table is built by the DAM; for HP-PB standard systems it provides a cross reference between the hardware address and software tables and addresses. The DAM creates a row in this table for each interface card that is bound by the system at bootup or reconfiguration. When the ZCOM subsystem is initialized, the LDM uses the information provided by TTGEN in the ZCOM interface table (INHPA field) to match the actual HPA value. If the LDM is able to find the matching HPA, it inserts the ZCOM interface table address in the HPA table.

Each row in the table has entries of the form (*zhpa_type*) shown in Table 3-44.

Table 3-44 HPA Table Entries

Field Name	Field Description	Field Type	Size (Bytes)
hpano	HPA value of this interface	long	8
resrvd1	Reserved for future use.	int	4

Table 3-44 HPA Table Entries (Continued)

Field Name	Field Description	Field Type	Size (Bytes)
resrvd2	Reserved for future use	int	4
card_type	The type of the ACC interface card	int	4
resrvd4	Reserved for future use	int	4
hlmpda	Pointer to lower manager's (DAM) PDA	pointer	8
hpaift	Pointer to ZCOM interface table	pointer	8
hstatus	Status of this entry	uns.short	2
hactive	Number of active requests	uns.short	2
sx25_brdn	Streams X.25 board number (used in Subnet ID)	uns.short	2
spare1	Reserved, not used	uns.short	2
dvr	Call entry for the interface driver	pointer	8
bus1_num	Bus converter position (U2) of card	int	4
bus_num	Bus number of card	int	4
slot_num	Slot address of the card	int	4
hw_path[]	Hardware path string for this card	char	40
hlock	IFT spinlock	struct	16

Note that for EISA ACC the slot number is assigned to PDA variable in the DAM.

HPANO - HPA value of I/F card (*zhpano_type*)

The HPA value is setup by the DAM when HP-UX creates an instance of the DAM for a particular Mux interface card. This field is used by the LDM to match the Mux interface cards defined in the TTGEN configuration file, to those actually configured in the hardware. The format of the field is shown in Table 3-45.

Table 3-45 HPA value of I/F card (HPANO)

63-32	31-28	27-18	17-14	13-12	11	10-6	5-2	1-0
All 1's (111...)	1111	Bus nmbr	Slot	Module	Page	Register Set	Reg No.	00

In the 32-bit kernel, the HPA value is a 32-bit value. Only the lower 32-bit are kept in this field. This field will also become a 32-bit field, followed by a 32-bit padding.

RESRVD1 - Reserved for future use.

RESRVD2 - Reserved for future use.

HLMPDA - Pointer to DAM PDA for I/F card

This is the address of the PDA for this Mux interface card and is setup by the DAM when it initializes a row for an interface card.

HPAIFT - Pointer to ZCOM Interface table for this interface

This field is initialized to NULL during bootup. It is set up by the LDM during ZCOM subsystem startup. It is used to retrieve the responsible IFT from the HPA value.

HSTATUS - Status of this entry

This field contains the status indicators for the HPA entry. The DAM sets up the status after the drivers recognize the hardware during HP-UX system boot-up. Currently, only one bit is used (bit 15, ZCOM_HSTATUS_ALIVE). Bit ZCOM_HSTATUS_ALIVE indicates that the Mux card is physically present and usable.

The LDM checks this status before it communicates with the DAM. If this bit is not set, the LDM will ignore the card. This feature is intended for the implementation of dynamic hardware reconfiguration in HP-UX.

HACTIVE - Number of active requests

This field contains the number of pending inter-driver requests from the LDM to DAM. A non-zero value indicates some LDM requests are being processed by the DAM, hence deallocation of the driver resources is not allowed (i.e., driver unbinding). This feature is intended for the implementation of dynamic hardware reconfiguration in HP-UX.

SX25_BRDN - Streams X.25 board number

This field is used internally by the driver for keeping track of the J2793A Streams X.25 assigned board number, used for this card in the SNID.

DVR - Call entry for the interface driver

This is a kernel function pointer to the DAM's primary entry point. It is for the LDM to request the DAM's services. It is set up by the DAM during system bootup.

BUS1_NUM - Bus converter position (U2) of the interface card

BUS_NUM - Bus number of the interface card

SLOT_NUM - Slot number of the card

These 3 fields are set up by the DAM during system bootup. It contains the hardware path of the associated interface card.

HW_ADDRESS - A string containing the hardware path to the card.

HLOCK - IFT spinlock

It is allocated by the DAM during system bootup. It is used to allow sharing resources in a Multi-Processor system. When a module requires exclusive access to a IFT, it must obtain this lock first.

SYS

The SYS structure (ZCOM system global information, *zsys_type*) contains software status information about the ZCOM subsystem. For example, it indicates whether ZCOM is running and when it was last started up. It is updated by the LDM based on requests from ZMON.

Table 3-46 SYS structure (ZCOM system global information, *zsys_type*)

Field Name	Field Description	Field Type	Size (Bytes)
STIME	Time when status last changed	uns.int	4
STATUS	Current system status	uns.int	4
DEBUG	Debug level, 0 = no debug	uns.int	4
MEMSIZE	Size of system memory	uns.int	4
MEMPTR	Pointer to system memory	pointer	8
SYSPTR	Pointer to ZCOM memory (aligned)	pointer	8
SYSSIZE	ZCOM system size (adjusted)	uns.int	4
NREQUEST	Number of active requests	uns.int	4
RTIMER	Reset request timer	uns.int	4
NSLOW	Number of slowdown processes	uns.int	4
ZSINFO	ZCOM system information buffer	struct	24

STIME - Time when status last changed

This field contains the time (in Epoch format) when the ZCOM subsystem changed to the current state. For example, in a running ZCOM subsystem, this field contains the time when the ZCOM subsystem became operative.

In addition to providing time information, this field is also used in the LDM sleeping mechanism. When a ZCOM application is awoken after waiting in the sleep state, it checks this field to determine whether the ZCOM subsystem has changed state (for example, from READY to

RESET state). If so, an error will be returned to the application program. This mechanism allows all sleeping ZCOM programs to be notified when the operational state of the ZCOM subsystem has changed.

STATUS - Current ZCOM system status

This field contains the current state of the ZCOM subsystem. The defined values are shown in Table 3-47.

Table 3-47 Current ZCOM system status

Name	Value (hex)	State
ZCOM_STAT_READY	0x84000001	Operative
ZCOM_STAT_BRESET	0x40000002	Being reset, going down
ZCOM_STAT_ABORT	0x20000003	Being reset, going down
ZCOM_STAT_RESET	0x10000004	Reset, not functioning
ZCOM_STAT_BOOT	0x10000005	System just booted up
ZCOM_STAT_RESTART	0x0C000006	Starting up
ZCOM_STAT_INIT	0x04000007	Memory initialized

The LDM uses the high order bits of this field as individual indicator to determine the status of the system as listed in Table 3-48.

Table 3-48 Current ZCOM system status (Higher Order Bits)

Name	Value (hex)	State
ZCOM_MASK_READY	0x80000000	Operative
ZCOM_MASK_BRESET	0x40000000	Being reset, going down
ZCOM_MASK_ABORT	0x20000000	Being reset, going down
ZCOM_MASK_RESET	0x10000000	Reset, not functioning
ZCOM_MASK_BSTART	0x08000000	System just booted up
ZCOM_MASK_INIT	0x04000000	Memory initialized

DEBUG - Debug level

This field is used by the LDM to produce driver debug logs for its processing. Some LDM modules have embedded debug codes, which check this field to log debug messages. The format of this field is shown in Table 3-49. The values of the bits in the field are described in Table 3-50.

Table 3-49 Debug Level Field Format

15-8	7	6	5-4	3-0
FID	MLOG	CLOG	UNUSED	LEVEL

Table 3-50 Debug Level Field Values

Bit	Name	Value (hex)	Meaning
LEVEL	ZCOM_DEBUG_LEVEL	0x0000000F	Debug level (0 means no debug, else debug enabled for all debug code this is of a level the same or lower than this value.)
CLOG	ZCOM_DEBUG_CLOG	0x00000010	If set, log debug message to system console.
MLOG	ZCOM_DEBUG_MLOG	0x00000020	If set, log debug messages to ZCOM message log.
FID	ZCOM_DEBUG_FID	0x0000FF00	Number to identify one of the LDM module to log debug messages (0 means all modules). The module ID is embedded in the source code. Refer to the source code to know the module ID of a particular file.

This field is maintained by the *zmon debug* and *zmon nodebug* commands. Because the debug logging increases the LDM overhead and is mainly for internal use, these commands are not recommended for users.

MEMPTR, MEMSIZE -Kernel memory size pointers

These are pointers to the size of kernel memory allocated by the LDM. The memory is used to hold ZCOM subsystem Memory Tables. In the current implementation, there is a fixed amount of memory defined in the kernel for use by the ZCOM subsystem, determined by the kernel ***zcom_mem_size*** tunable parameter defined at kernel build time. Hence, these fields are assigned to that part of memory.

SYSPTR, SYSSIZE - ZCOM system size (adjusted) and pointer.

These are the pointers to the size of the kernel memory to be used by the ZCOM subsystem. The ZCOM subsystem memory requires 64-byte alignment, and the system may not use up all allocated memory. These two fields indicate the actual location and size of the ZCOM memory.

NREQUEST - Number of active requests

This field contains the number of ZCOM application requests in progress. The LDM increments content in this field when it receives a new request, and decrements it when a request completes. During ZCOM subsystem shutdown, the LDM checks this value for pending application requests. If there is any pending requests, the LDM waits (for 5 seconds) to give the requests time to complete before forcing the ZCOM subsystem down.

This field is intended for the implementation of a smooth (i.e., graceful) ZCOM subsystem shutdown mechanism.

RTIMER - Reset request timer

This is the timer used by the LDM to wait during a ZCOM subsystem shutdown. The LDM will not shutdown the ZCOM subsystem immediately if there are pending application requests. Instead, it sets this timer to a non-zero value and goes dormant. When the LDM timer module is activated, it decrements this field and wakes up the waiting LDM when it reaches zero.

This field is used with NREQUEST to provide a waiting deferred shutdown mechanism.

NSLOW - Number of slowdown processes (* obsolete *)

This field is obsolete. Not used.

ZSINFO - ZCOM system information buffer

This is a data structure set up by the LDM when ZCOM subsystem is started up. It contains information to be passed back to an application via *zinit*. Refer to the definition of *zsinfo_type* for its layout.

HP

The HP (Pointer to ZCOM system header) structure is a pointer to ZCOM subsystem header. It is set up by the LDM during ZCOM subsystem startup. The ZCOM subsystem memory tables are accessed through this pointer.

LINFO

The LINFO structure (Lock Information, *struct zlinfo*) contains the ZCOM locks used globally in the ZCOM subsystem. It is initialized by the LDM during system bootup. The locks are used to provide exclusive access to the sharable data structures in the ZCOM subsystem. The fields are described in Table 3-51.

Table 3-51 LINFO Structure Fields

Field Name	Field Description	Field Type	Size (Bytes)
ZC_GLOCK	Global spinlock	struct	16
ZC_MLOCK	Memory spinlock	struct	16
ZC_RLOCK	Remote ZLU spinlock	struct	16
ZC_SLOCK	Super IFT spinlock	struct	16

ZC_GLOCK - Global spinlock

This is the highest lock order in the ZCOM subsystem. It protects the ZCOM system header *zheader_type* in global memory (with some exception, see ZC_RLOCK below), and the LDM static data such as the interface request queue and ZCOM message log buffer.

ZC_MLOCK - Memory pool spinlock

This protects the ZCOM buffer pool. Used in buffer management routines, such as buffer allocation and buffer freeing.

ZC_RLOCK - Remote ZLU spinlock

This lock protects the following ZCOM system header data fields. They are connected with remote nodes or remote API processing:

hrdataq, hpntbl, hndpid, hndsig, hndmax, hnidle, hnhigh, hnlow, hnwait

ZC_SLOCK - Super IFT spinlock

This lock protects against deadlock situations while multiple IFTs need to be access simultaneously. A piece of code must obtain this lock first if it is going to lock multiple IFTs.

Each of the above fields is a data structure (*zlock_t*) with the data fields shown in Table 3-52.

Table 3-52 Super IFT spinlock (ZC_SLOCK) Data Structure

Field Name	Field Description	Field Type	Size (Bytes)
LOCKP	Pointer to kernel spinlock structure	pointer	8
REFCOUNT	Lock reference count	int	4
SPARE1	Reserved, not used	int	4

LOCKP - Pointer to kernel spinlock structure

This is a pointer to the spinlock structure allocated dynamically from the kernel. It is used for the locking mechanism in the kernel.

REFCOUNT - Lock reference count

This is to support locks that could be acquired recursively in the ZCOM subsystem. Every time when the ZCOM lock is acquired, this count is incremented. When the ZCOM lock is released, this count is decremented. Only when this count reaches zero, the kernel lock (via the LOCKP pointer) is released.

4 ZCOM C I/F Library Routines

Introduction

The ZCOM C interface library routines provide an application program interface to the ZCOM subsystem. These enable the application programmer to access to the ZCOM subsystem, transfer data, configure the system, check on the state of the ZCOM subsystem, etc.

Applications should link in the archive library *libzcom_c.a* or the shared library *libzcom_c.sl* with their program to use these functions. Two sets of libraries are provided: the 32-bit libraries support 32-bit programs, portable across both PA-RISC 1.1 and PA-RISC 2.0 hardware platforms, while the 64-bit libraries **only** support 64-bit programs running only on PA-RISC 2.0 64-bit hardware (wide mode). The 32/64-bit libraries have the same name, but the location is different; */opt/acc/lib* for the 32-bit library, and */opt/acc/lib/pa20_64* for the 64-bit library. The library location can be specified by preceding the -l option to *cc* or *ld* with "-L<dir>".

Note that the HP-UX 11.0 (or later) operating system supports cross platform development. Regardless of the hardware model, the compiler option +DA1.1 (or +DAportable) will generate a 32-bit program, while the option +DD64 (or +DA2.0W) will generate a 64-bit program.

All the ZCOM library routines are thread-safe and can be called from multi-threaded applications. They are implemented using the POSIX (1003.1c) kernel threads API package, therefore it is necessary to include the thread library *libpthread.a* after the ZCOM library. Refer to the man page *pthread(3T)* for more details on POSIX threads.

The following are the suggested compiler and linker options for ACC application program using the ZCOM C library:

```
32-bit program: +DA1.1 -L/opt/acc/lib -lzcom_c -lpthread

64-bit program: +DD64 -L/opt/acc/lib/pa20_64 -lzcom_c -lpthread
```

NOTE All "Include" files formerly under */usr/include/zcom* have been relocated to */opt/acc/include/zcom*. Users should add "`-I/opt/acc/include`" to their "make" files, to direct the preprocessor to also include any files in the directory specified.

The following is a list of ZCOM C interface library routines:

Initialization call

zinit	-ZCOM library access initialization

Interface configuration calls

zport	-Datacomm port configuration
zconfig	-Dynamic system configuration

ZLU configuration calls

zclos	-Delete program input ZLU
zmapr	-Set up ZLU mapping configuration
zopen	-Create ZLU program input queue
zluopen	-Open a ZLU for file system access
zset_rcvr	-Set up a program ZLU as receiver for a terminal ZLU
zevent_rcvr	-Set up a program ZLU as receiver for system event messages
zsetql	-Set buffer queue limit
ztimr	-Enable/disable ZLU timer

Data storage allocation calls

zltqueue	-Logical data queue allocation
zltstore	-Logical data storage allocation

ZLU data transfer calls

ltqdget	-Read data buffer in terminal data queue
ltqdput	-Store data buffer in terminal data queue
zcntl	-Send control message to ZLU
zpeek	-Read data without disturbing queue
zqmve	-Move message from one ZLU to another
zread	-Read from ZLU

zsend	-Send data buffer to ZLU

ZLU information calls

ltfind	-Find specific logical terminal in a group
ptfind	-Find specific physical terminal in a group
zinfo	-Get ZCOM table information
zname	-Find ZLU number from ZLU name
zqsze	-Read ZLU input queue size
zget_shrcvr_list	-Retreive a list of all the current shared receivers

Utility routines

getdevice	-Return ZCOM device definitions
makezluname	-Make a ZLU name with tty name suffix
zcomerror	-Return a formatted ZCOM error message
zcomlname	-Set up originator name for log messages
zcomlog	-Add a message to the ZCOM error log file
zcomstatus	-Return a formatted ZCOM status or error message
zltmg	-Move logical terminals between groups
zltup	-Update logical terminal table
zptup	-Update physical terminal table
zrntimer	-Set ZCOM remote node access timeout

ZLU Definition

The *zlu* field of a <*zaddr_type*> parameter specifies the ZLU address. The *checksum* field of a <*zaddr_type*> parameter is a checksum used to verify the address of the message. All routine calls that specify '(with checksum)' in the ZLU definition must have both fields defined correctly for successful processing.

The checksum minimizes the possibility of messages being accidentally misdirected. The checksum is calculated based on the type and name of the ZLU, hence the checksum does not change even when the system is restarted. If the program that opened the ZLU is killed, it may reopen the same ZLU by referencing it by ZLU name and the same checksum will be generated. This allows any other message sources to deliver data to a restarted ZLU without any impact.

The *node* field of a <*zaddr_type*> parameter must contain a valid node number or zero if it is an input parameter. However, if it is a return parameter, such as in the *zopen* call, then the *node* number returned is the local node as defined in the "configuration file". Note that specifying a node number of zero is the same as specifying the local node number. If a remote node is specified, it must be defined in the *Node-Definition* section of the TTGEN configuration file. Refer to the *ACC Utilities Guide* for more information about the TTGEN configuration or the *ttgen(1)* man page file.

Man Pages

The C-I/F man pages are provided in this section. They are in alphabetical order in similar format as the online man pages.

- getdevice.3x
- ltfind.3x
- ltqdget.3x
- ltqdput.3x
- makezluname.3x
- ptfind.3x
- zclos.3x
- zcntl.3x
- zcomerror.3x
- zcomlname.3x
- zcomlog.3x
- zcomstatus.3x
- zconfig.3x
- zevent_rcvr.3x
- zget_shrcvr_list.3x
- zinfo.3x
- zinit.3x
- zltmg.3x
- zltmx.3x
- zltqueue.3x
- zltstore.3x
- zltup.3x
- zluopen.3x
- zmapr.3x
- zname.3x
- zopen.3x
- zpeek.3x
- zport.3x
- zptup.3x
- zqmve.3x
- zqsze.3x
- zread.3x
- zrntimer.3x
- zsend.3x
- zset_rcvr.3x
- zsetql.3x
- ztimr.3x

GETDEVICE (3X)

NAME

getdevice – Read ZCOM device definitions

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C
(recommended) */
     int getdevice (dfile, dpp)
     char                  *dfile;
zdev_type        **dpp;
```

DESCRIPTION

Routine *getdevice* reads the ZCOM device file specified by the *dfile* parameter and loads the device definition entries into a memory table. The pointer to the device definition table is returned (see *zdev type* definition below).

If *dfile* is NULL, it uses the default */opt/acc/cfg/zcomdevice*. The ZCOM device file is a binary file created by the ZDGEN program, from a user-customizable ASCII file (usually */opt/acc/cfg/zcomdevice.txt*). Refer to the section on *zdgen* in the *Multiprotocol ACC Utilities Reference Guide* for more details.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the NOTES section below for more information on using this routine in a multi-threaded application.

PARAMETERS

dfile	Pointer to the full path name of a binary *zdgen* device file. That is, the output file from the *zdgen* program. If a NULL is passed, it defaults to the file specified by the symbol ZCOM_DEVICE_FILE), which is currently defined as */opt/acc7cfg/zcomdevice* in the include file */opt/acc/include/zcom/zcomsys.h*
dpp (**Return param**)	Pointer to a data structure containing the full device table. The device table is an array of zdev_type elements.

RETURN VALUE

Routine *getdevice*, when successful, returns a **non-negative** value, indicating the number of device types. That is, it returns the number of zdev_type entries in the device table. Otherwise, it returns a **negative** value, and the user should check errno to determine why *getdevice* failed.

NOTES

1. On every call to *getdevice*, *malloc*(3C) is called to allocate dynamic memory for the returned device table. In case of error, this function will release the allocated memory internally to avoid memory leak. On successful return, the caller is responsible for freeing the allocated memory when it is no longer needed.
2. In a multi-threaded application, if thread cancellation occurs while calling this function, the allocated memory will be released before the thread terminates. Once this function returns, the calling thread is responsible for freeing the allocated memory during thread cancellation.
3. The following is the *zdev_type* data structure definition (from /opt/acc/include/zcom/zcomsys.h).

```
/*-- ZCOM Device Entries --*/
typedef struct {
    char dvname[16]; /* Device name */
    uint16 dptype;    /* Physical type (i.e., protocol number)
*/
    uint16 dltype;   /* Logical type (i.e., screen type) */
    uint16 dalen     /* Poll/Select address length (0,1,2) */
    uint16 damin;    /* Lower bound of address */
    uint16 damax;    /* Upper bound of address */
    uint16 doptn;    /* Protocol options */
    uint16 dlflag;   /* Logical terminal flags */
```

```
    char dpname[8];  /* Protocol module name */
} zdev_type;
```

EXAMPLE

```
#include        <zcom/zcomsys.h>
#include        <zcom/zcomcall.h>

int32        ierr;
char         *dfile;
zdev_type    *dp;

ierr = getdevice (dfile, &dp);
if (ierr < 0) {
/* (error; user to check errno) */
}
else {
/* (success; ierr indicates number of device types) */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note: This must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).
/opt/acc/cfg/zcomdevice.txt	User-customizable ASCII file used by ZDGEN to create the binary ZCOM device file, */opt/acc/cfg/zcomdevice*. Refer to the section on ZDGEN in the *Multiprotocol ACC Utilities Reference Guide* for more details.

/opt/acc/cfg/zcomdevice	Binary ZCOM device file created by ZDGEN, using the user-customizable ASCII file, */opt/acc/cfg/zcomdevice.txt*. Refer to the section on ZDGEN in the *Multiprotocol ACC Utilities Reference Guide* for more details.

SEE ALSO *zdgen(1), malloc(3C)*

LTFIND (3X)

NAME

ltfind – Find specific logical terminal in a group

SYNOPSIS

```
#include         <zcom/zcomsys.h>
#include         <zcom/zcomcall.h> /* if compiled with ANSI C
(recommended) */

int32 ltfind (zap, laddr, llen, ibuf, len)

zaddr_type       *zap;
char             *laddr;
int32            llen;
char             *ibuf;
int32            len;
```

DESCRIPTION

Routine *ltfind* searches through the group linkage in the logical terminal table, for the logical terminal with the specified logical address.

Group linkages are set up by TTGEN to form a circular list of logical terminals that are associated with the same physical terminal table. The linkages and logical addresses are defined using the PTERM and LTERM statements in the TTGEN configuration file.

Typically, these linkages are used in a multiplexing environment: a lower level protocol receives multiplexed messages, which contain ID codes for a higher level protocol to further differentiate the data messages. In such cases, the logical terminals may have unique ID codes as their logical addresses, and *ltfind* is used to locate a logical terminal ZLU with a specified ID code.

Normally, the logical addresses (maximum of 10-bytes) of a group of logical terminals should be unique within that group.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” options to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address of a terminal in the multiplexed group.
laddr	Logical address with which to compare.
llen	Length in bytes of laddr. A maximum of **10** bytes/characters of *laddr* will be used, even if a number greater than 10 is specified in *llen*.
ibuf **(Return Param)**	Logical terminal table. Note that if the logical terminal is not found, *ibuf* will not contain valid data.
len	Length in bytes of *ibuf*.

RETURN VALUE

Routine *ltfind* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings. If the logical terminal is not found, ZENOTM (-75) is returned.

NOTES

The returned logical terminal table consists of the basic table and the logical terminal table extension (the extension size is configurable in the TTGEN configuration file). The size of this buffer should be large enough to hold the information required from the LTT. An example of defining a buffer of sufficient size would be:

```
struct {
   zltt_type   sys;
   char         ext[ZCOM_MAXLSIZE];  /* Max allowable extension */
ltt;
```

For example, to access the fields:

ltt.sys.ltname	gives the terminal name
ltt.ext[0]	gives the 1st byte in table extension

This allows *&ltt* to be used instead of *ibuf* in the *ltfind* call.

EXAMPLE

```
#include        <zcom/zcomsys.h>
#include        <zcom/zcomcall.h>

int32      ierr;
zaddr_type zaddr;
char       laddr[10];
int32      llen;
char       ibuf[size]; /* where size is user-determined; must be >= len */
int32      len;

        if (ierr = ltfind (&zaddr, laddr, llen, ibuf, len)) {
           /* error return code */
}
else {
           /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zdgen(1), malloc(3C)

LTQDGET (3X)

NAME

ltqdget – Retrieve data buffer in terminal data queue

SYNOPSIS

```
#include       <zcom/zcomsys.h>
#include       <zcom/zcomcall.h> /* if compiled with ANSI C
(recommended) */

int32 ltqdget (zap, queue, ibuf, len, rlenp)

zaddr_type     *zap;
uint32         queue;
char           *ibuf;
int32          len;
int32          *rlenp;
```

DESCRIPTION

Routine *ltqdget* retrieves the data message stored in the data queue of the logical terminal tables. It returns the first message in the specified queue and releases the buffers back to the ZCOM subsystem free data buffer pool. See the man page for *ltqdput* on how to store a message in the data queue.

The data queue is designed to store a relatively large or unpredictable amount of terminal data in a FIFO order. When compared to logical data storage (refer to *zltstore* and *zltup*), the data queues allow terminal data to be stored dynamically, that is, memory is used only when data is stored. However, it is penalized by incurring more processing overhead in data update and retrieval. In addition, access is restricted to a FIFO order.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.

async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address of a terminal from which to get the data.
queue	Terminal data queue number (specifies which data queue is to be used). 0 - Sub-packet holding queue (reserved) 1 - Data holding queue A 2 - Data holding queue B 3 - Data holding queue C 4 - Data holding queue D
ibuf (Return Param)	On a successful return, the data is placed into this buffer.
len	Length in bytes of *ibuf*.
rlenp (Return Param)	Actual number of bytes returned in *ibuf*.

NOTES

Normally, the data queue number used should be allocated by ZMON (see the *zltqueue* man page for how to get a data queue number). There is no queue ownership validation within this routine; it is up to the user to make sure that the queue usage is valid and consistent.

RETURN VALUE

Routine *ltqdget* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

If the specified queue is empty, then the **ZENODATA** error (**-24**) is returned and *ibuf* contains invalid data.

EXAMPLE

```
#include    <zcom/zcomsys.h>
#include    <zcom/zcomcall.h>

int32        ierr;
zaddr_type   zaddr;
uint32       queue;
char         ibuf[size];  /*Where size is user-determined; must be
>= len */
int32        len;
int32        rlen;

if (ierr = ltqdget (&zaddr, queue, ibuf, len, &rlen)) {
```

```
      /* error return code */
}
else {
      /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

ltqdput(3X), zltqueue(3X), zltstore(3X), zltup(3X)

LTQDPUT (3X)

NAME

ltqdput – Store data buffer in terminal data queue

SYNOPSIS

```
#include <zcom/zcomsys.h>
#include <zcom/zcomcall.h> /* if compiled with ANSI C (recommended)
*/

int32 ltqdput (zap, queue, ibuf, len)

zaddr_type  *zap;
uint32      queue;
char        *ibuf;
int32       len;
```

DESCRIPTION

Routine *ltqdput* stores a data message in the data queue of the logical terminal tables. It links the message to the end of the specified queue. Retrieval occurs on a First-In-First-Out basis. See the man page for *ltqdget* on how to retrieve a message in the data queue.

The data queue is designed to store a relatively large or unpredictable amount of terminal data in FIFO order. When compared to logical data storage (refer to *zltstore* and *zltup*), the data queues allow terminal data to be stored dynamically, that is, memory is used only when data is stored. However, it is penalized by incurring more processing overhead during data update and retrieval. In addition, access is restricted to a FIFO order.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzx25 -lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread’s cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.

async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address of the terminal ZLU to store data into.
queue	Terminal data queue number (specifies which data queue is to be used): 0 - Sub-packet holding queue (reserved) 1 - Data holding queue A 2 - Data holding queue B 3 - Data holding queue C 4 - Data holding queue D
ibuf	Data buffer to be stored.
len	Length in bytes of *ibuf*.

NOTES

Normally, the data queue number used should be allocated by ZMON (see the man page for *zltqueue* on how to get a data queue number). There is no queue ownership validation within this routine; it is up to the user to make sure that the queue usage is valid and consistent.

RETURN VALUE

Routine *ltqdput* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include        <zcom/zcomsys.h>
#include        <zcom/zcomcall.h>

int32             ierr;
zaddr_type        zaddr;
uint32            queue;
char              ibuf[size];
int32             len;

if (ierr = ltqdput (&zaddr, queue, ibuf, len)) {
      /* error return code */
}
else {
      /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

ltqdget(3X), zltqueue(3X), zltstore(3X), zltup(3X)

MAKEZLUNAME (3X)

NAME

makezluname – Make a ZLU name with TTY name suffix

SYNOPSIS

```
#include <zcom/zcomsys.h>
#include <zcom/zcomcall.h> /* if compiled with ANSI C (recommended)
*/

char *makezluname (name)
char *name;
```

DESCRIPTION

Routine *makezluname* constructs a ZLU name by merging the user-supplied *name* with the TTY suffix. The returned name is 7 characters long plus a terminating '\0'. This name may be provided in a call to *zopen* to open or create a program ZLU with a name that is associated with the TTY of the calling process. Note that the *zopen* call assigns a label (name) to each program ZLU and does allow two program ZLUs to be created with the same name. This allows a program to be run from different terminals without referring to the same program ZLU.

This routine calls *ttyname* to retrieve the TTY name of the current process. From the returned TTY *name*, the prefix "tty" is stripped and then appended to the user-supplied name to form a ZLU *name*. Note that part of the user-supplied *name* and TTY *name* may be truncated to ensure that the returned ZLU name fits in an 8-byte buffer.

For example, if the specified name is "MYZLU", the following shows the returned ZLU name for different terminals.

TTY Name	Returned Name	Comment
tty01	MYZLU01	Only TTY suffix is used.
tty0p1	MYZL0p1	"U" is dropped to fix TTY suffix.
tty04p12	MY04p12	Maximum of 5 characters from TTY suffix.
tty04p12i	MY4p12i	Only 5 of 6 characters from TTY suffix.
term	MYZterm	Full TTY name is used.

TTY Name	Returned Name	Comment
console	MYnsole	Trailing part of TTY name is used.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the WARNING section below for more information on using this routine in a multi-threaded application.

PARAMETERS

name Name to be merged with TTY suffix.

RETURN VALUE

Routine *makezluname* returns a null-terminated **character string**, which consists of the user-supplied *name* parameter merged with the TTY name suffix.

WARNING

If two different programs call this routine with the same *name* parameter and from the same TTY, the same ZLU name will be returned.This implies that if multiple copies of a program are run from the same TTY, they will all get the same ZLU name. If they are run from different terminals, different ZLU names will be returned.

For multi-threaded application, since all threads will be from the same TTY, if the threads use the same *name* parameter, the same ZLU name will be returned.

This routine uses an internal static buffer for temporary storage of the returned name. The buffer is overwritten by each call.

EXAMPLE

```
#include <zcom/zcomsys.h>
#include <zcom/zcomcall.h>

char     *zluname;   /* Returned ZLU name string */
char     name[8];

zluname = makezluname (name);
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zopen(3x), ttyname(3c)

PTFIND (3X)

NAME

ptfind – Find a physical terminal within a group

SYNOPSIS

```
#include         <zcom/zcomsys.h>
#include         <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 ptfind (zap, zskeyp, ibuf, len)

zaddr_type       *zap;
zskey_type       *zskeyp;
zptt_type        *zptt;
int32            len;
```

DESCRIPTION

Routine *ptfind* provides special methods of accessing terminals on the same MUX and port. The zap->zlu parameter is the ZLU of a terminal on a particular MUX and port.

Physical terminals that are associated with the same port are called a physical terminal group. They are linked together to form a circular list via the *ptpzlk* and *ptlist* fields in the physical terminal table. This routine provides a method to efficiently search through a group.

This routine provides three methods of finding a particular physical terminal table entry (and relative terminal number) on the same MUX and port as the ZLU given (*zap->zlu*) in the first parameter.

Method 1: Retrieve the terminal associated with a specific MUX, port, and terminal number.

In the ZCOM subsystem, every physical terminal belongs to a single MUX and PORT and is assigned a TERM (terminal) number, which is unique within that port. Hence, by specifying these three numbers, a single physical terminal table may be retrieved. The TERM number is assigned by TTGEN during configuration file compilation and kept in the *ptterm* field of the physical terminal table.

This method is used usually when the MUX, PORT, and TERM are known. An example is in retrieving a previously accessed physical terminal table.

Method 2: Retrieve the next terminal whose table has a sequence of matching bytes.

This method compares the contents of the physical terminal tables within a group, and locates the one that matches a byte sequence. In addition to the data buffer, table location and number of bytes to be compared, the caller also specifies a terminal ZLU within the group to be searched.

In order to make it easier to handle multiple matches in a group, the search starts from the next terminal following the specified ZLU. Hence, a returned terminal ZLU may be used in the next *ptfind* call to locate the next matching terminal. In this case, the whole group is searched if a terminal is returned the second time.

Method 3: Retrieve the next terminal whose table has a sequence of matching bit patterns.

This mode is similar to Method 2 except that the caller also specifies a mask buffer, which is used to mask some unused bits in the physical terminal table before comparing with the specified data.

The mask buffer is AND'ed with the table content before comparison. Hence, the mask buffer should contain zeros for those unused bits and ones for those bits to be compared. The data buffer should have zeros for those unused bits, 0 or 1 for the bits to be compared.

Similarly, the search starts from the next terminal after the specified ZLU. (In fact, method 2 may be treated as a special case of this, if the mask buffer contains all ones.)

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address of the terminal. It can be any terminal ZLU within the group to be searched. If using method 1, only its MUX number is used (its port number, hence its group, may be different from the target terminal).
zskeyp	*ptfind* search key. The search key is set up by the caller program before this routine is called. It consists of a search mode, which indicates the method to be used and the data (mode dependent) used in searching. If bit ZCOM_PTFIND_NAVL (0x8000) is set, *ptfind* searches the non-available terminals as well. When using this function in a 32-bit program that is running on a 64-bit system, you must be especially careful to zero out the zskeyp structure before using it. Otherwise, your application may experience ZCOM error -84 (ZEBPTR), “User buffer access error”.
zptt **(Return Param)**	A pointer to a buffer into which the ZCOM physical terminal table data will be placed. Note that if the physical terminal is not found (i.e., *ierr* != 0), then no data is moved into *zptt*.
len	Length in bytes of *zptt* buffer.

RETURN VALUE

Routine *ptfind* returns **0** if successful. Otherwise, if the physical terminal is not found, **ZENOTM (-75)** is returned.

EXAMPLE

```
#include        <zcom/zcomsys.h>
#include        <zcom/zcomcall.h>

int32           ierr;
zaddr_type      zaddr;
zskey_type      zskey;
zptt_type       zptt;
int32           len;

if (ierr = ptfind (&zaddr,  &zskey, &zptt, len)) {
           /* error return code */
}
```

```
else {
            /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZCLOS (3X)

NAME

zclos - Delete program input ZLU

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32       zclos (zap)

zaddr_type  *zap;
```

DESCRIPTION

Routine *zclos* closes the program ZLU set up by the *zopen* call and returns the ZLU back to the system for reuse. This call should normally always occur before a program terminates. If the program ZLU is not closed, then the ZCOM subsystem will continue to hold all of the resources associated with the ZLU and may add messages to the input queue. This feature allows a program to open a ZLU on another program's behalf, or to be killed and rerun without losing any queued messages. It is the programmer's responsibility to ensure that all unnecessary program ZLUs are closed before the program terminates.

If *zclos* is called to close a program's primary ZLU, the program will no longer have a primary ZLU. Usually, a program should open another ZLU as the primary ZLU because some APIs may return an error if the primary ZLU does not exist (e.g. *zsend*(3X)).

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.

async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

za p	ZCOM address to be returned to the system.

RETURN VALUE

Routine *zclos* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include        <zcom/zcomsys.h>
#include        <zcom/zcomcall.h>

int32           ierr;
zaddr_type     zaddr;

if (ierr = zclos (&zaddr)) {
           /* error return code */
}
else {
           /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zopen(3X)

ZCNTL (3X)

NAME

zcntl – Send control message to terminal ZLU

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zcntl (zap, mode, rcode, mhp, ibuf, len, rstat)

zaddr_type      *zap;
uint32          mode;
uint32          rcode;
zmhd_type       *mhp;
char            *ibuf;
int32           len
int32           *rstat;
```

DESCRIPTION

Routine *zcntl* is used to change the state of a terminal or to send a protocol dependent control message to a terminal ZLU. The routine *zsend*(3X) should be used for sending normal data messages to a terminal.

The terminal must be disabled first if configuration parameters are to be changed (e.g. *rcode*=6). As in *zsend*(3X), the *mode* parameter is used to specify how the call should be completed and for selecting the priority of the request. See under *zsend*(3X) for more information on its usage.

The response returned from zcntl is through the primary ZLU unless a different source ZLU is specified in the header (*mhp->mid.mzsrce*).

Note that a program must call zopen() to allocate a primary or auxiliary input ZLU before calling the zsend() routine. This must be done even if the program does not need an input ZLU, for example, if it is not doing any zread() calls and is only doing zsend() calls with mode = ZcMODE_NO_WAIT (0) (send no-wait) where it does not care about the return status.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the NOTES section for more information on using this routine in a multi-threaded application.

PARAMETERS

zap Pointer to terminal ZCOM address. The destination where the control message is sent.

mode Completion control:

ZMODE_NO_WAIT	(0) - Send no wait
ZMODE_RPT_ERRS	(1) - Send no wait, report errors
ZMODE_DEF_STATUS	(2) - Send no wait, definite status
ZMODE_RPT_ERRS_WBUF	(5) - Send no wait, report errors with buffer
ZMODE_DEF_STATUS_WBUF	(7) - Send no wait, definite status and buffer
ZMODE_WAIT	(8) - Send and await status in *rstat*

Modes **5** and **7** are only meaningful when rcode is **3**.

Some bits (when set) in the mode parameter can cause *zcntl* to behave differently. Refer to the NOTES section in *zsend*(3X) for details.

rcode — Control request code:

ZCOM_MRQCODE_CNTWR	(3) - Control write (length in *len*)
ZCOM_MRQCODE_TERM	(6) - Set terminal parameters
ZCOM_MRQCODE_ENB	(7) - Enable terminal
ZCOM_MRQCODE_DSB	(8) - Disable terminal
ZCOM_MRQCODE_ACT	(9) - Activate polling
ZCOM_MRQCODE_DEA	(10) - Deactivate pollin

To operate normally, a terminal ZLU must be enabled (*rcode*=ZCOM_MRQCODE_ENB) and activated (*rcode*=ZCOM_MRQCODE_ACT). If the terminal ZLU is deactivated (*rcode*=ZCOM_MRQCODE_DEA), the ZCOM subsystem cannot receive incoming messages from that terminal (used for flow-control). If it is disabled (*rcode*=ZCOM_MRQCODE_DSB), data messages can neither be sent nor received from the terminal.

Request ZCOM_MRQCODE_TERM is used to configure a terminal with new parameters, which are protocol dependent. Control write request ZCOM_MRQCODE_CNTWR provides a means of passing protocol specific control information to a terminal. These two requests require setting up the *ibuf* and *len* parameters.

mhp — Pointer to message header. It contains some data fields to be sent with the data message. See the NOTES section in *zsend*(3X) for what fields must be set up by the caller. If NULL is specified, default values are used.

For its structure, refer to the subsection on 'Message Header' in the *Multi-protocol ACC Programmers' Reference Guide*.

ibuf, len — If *rcode*=ZCOM_MRQCODE_CNTWR, ibuf contains the control data for the protocol, and len indicates the length of the control data. If *rcode*=ZCOM_MRQCODE_TERM, *ibuf* contains the configuration data for the terminal as described below, and the *len* indicates the total length of the configuration data. For other values of *rcode*, *ibuf* and *len* are not used.

rstat **(Return Param)** — Return status. Only used for requests issued with 'wait', i.e. *mode* = ZMODE_WAIT. NULL may be used if the return status is not required.

NOTES

1. File */opt/acc/include/zcom/zcomsys.h* contains the symbolic names for *rcode* (not all of them are applicable to *zcntl*):

```
/* Message request code values applicable to zcntl() */
#define ZCOM_MRQCODE_CNTWR    3 /* Terminal write */
#define ZCOM_MRQCODE_STATUS   5 /* Unsolicited status change */
#define ZCOM_MRQCODE_TERM     6 /* Set terminal parameters */
#define ZCOM_MRQCODE_ENB      7 /* Terminal enable */
#define ZCOM_MRQCODE_DSB      8 /* Terminal disable */
#define ZCOM_MRQCODE_ACT      9 /* Terminal activate */
#define ZCOM_MRQCODE_DEA      10 /* Terminal deactivate */
```

2. The result for a *zcntl* request is returned in the *rstat* parameter if *mode*=8, or may be returned in a response message via *zread*(3X) if another non-zero *mode* is specified. Refer to the NOTES section in *zsend*(3X) for more information on the *mode* parameter. See *zread*(3X) for more details on response messages.

3. For *rcode*=ZCOM_MRQCODE_TERM (Set terminal parameters), the terminal configuration data in *ibuf* must be set as follows. Refer to */opt/acc/include/zcom/zcomsys.h* for these data structures:

   For any ACC card except an i960 4-channel ACC card:

   Data structure: *ztrq_type*
   Must set RTYP bit = 0.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	RTYP	0	Terminal Type						-	-	-	-	-	Port #		
2	Group Poll Code								Device Poll Code							
4	Group Select Code								Device Select Code							

6	Terminal Option Word

For any i960 4-channel ACC card:

Data structure: *ztrq2_type*
Must set RTYP bit = 1. This usually means setting the *ztrq2_type.tmreqt* field to 0x80.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	RTYP	Ext. Req. Type							Terminal Type							
2	Port Number								Subchannel Number							
4	Group Poll Code								Device Poll Code							
6	Group Select Code								Device Poll Code							
8	Terminal Option Word															
10	Special Terminal Configuration (Variable Length, max. 6 bytes)															

The terminal type relates a protocol to the specific terminal (the device name in a TTGEN configuration file maps to a terminal type for each terminal). The terminal port must be the same as the original port. To move a terminal from one port to another, use *zconfig*(3X). The special terminal configuration is protocol dependent, and corresponds to the field *spec_cfg* of the physical terminal table (see *zptt_type* in */opt/acc/include/zcom/zcomsys.h*). Currently, a maximum of 6-byte is allowed. Configuration data larger than this is silently truncated.

4. The tag parameters (*mhp->mid.mtagw1* and *mtagw2*) may be used in *zcntl* for identifying the origin of a response when returned from *zread*(3X). They may also contain codes indicating the specific action to be taken on completion of the request.

RETURN VALUE

Routine *zcntl* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

If *mode* 8 is used and *zcntl* returns with zero, *rstat* contains the return status: **0** means successful, a **non-zero** value means there was a problem with the control request. *zcomstatus*(3X) may be used to retrieve a status message, using *rcode* and *rstat*.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32        ierr;
zaddr_type   zaddr;
uint32       mode;
uint32       rcode;
zmhd_type    zmhd;
char         ibuf[size]; /* where size is user-determined; must be >= len */
int32        len;
int32        waitstat;

if (ierr = zcntl (&zaddr, mode, rcode, &zmhd, ibuf, len, &waitstat)) {
    /* error return code */
}
else {
    /* good return code */
}

/* For request ZCOM_MRQCODE_TERM for i960 card (e.g. Z7300A) */
ztrq2_type  ztrq2;
char             spec_cfg[6];

ztrq2.tmreqt = 0x80;
/* ... setup other ztrq2 fields and spec_cfg[], max 6 bytes */
memcpy(ibuf, &ztrq2, sizeof(ztrq2));
memcpy(ibuf + sizeof(ztrq2), spec_cfg, 6);
rcode = ZCOM_MRQCODE_TERM;
len = sizeof(ztrq2) + 6;
if (ierr = zcntl (&zaddr, mode, rcode, &zmhd, ibuf, len, &waitstat)) {
     /* error return code */
}
else {
     /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.

/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO zcomstatus(3X), zconfig(3X), zread(3X), zsend(3X), zopen(3X).

ZCOMERROR (3X)

NAME

zcomerror – Return a formatted ZCOM error message

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

char      *zcomerror(err)
int32     err;
```

Description

Routine *zcomerror* returns a character string containing error text associated with an error number returned through use of the ZCOM library calls. It works in a similar fashion to *zcomstatus*, however, it has only one parameter passed in the call.

The return character string can be up to 31 bytes in length, plus a terminating zero.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

not a cancellation point	Thread cancellation will not occur when a thread is calling this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

err Error number.

NOTES

This routine reads a message file for the error message text. It uses the file zerrmsg.msg in /opt/acc/msg (ZCOM_MESSAGE_PATH). If not found, it uses the default message file *default.msg*.

RETURN VALUE

Routine *zcomerror* returns a character string corresponding to the error number passed in as a parameter (*err*).

EXAMPLE

```
#include    <zcom/zcomsys.h>
#include    <zcom/zcomcall.h>

char        *errmsg;  /* Returned message string */
int32       err;

errmsg = zcomerror (err);
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).
/opt/acc/msg/zstatmsg.msg	ZCOM status messages.
/opt/acc/msg/default.msg	Master message text file.

ZCOMLNAME (3X)

NAME

zcomlname – Set up originator name for log messages

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

void  zcomlname (name)
char     *name;
```

DESCRIPTION

Routine *zcomlname* sets up the originator (program) name to be used by the ZCOM message log system. Subsequent calls to *zcomlog* will log messages with this specified program name. Refer to *zcomlog(3x)* for more information on log messages.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

not a cancellation point	Thread cancellation will not occur when a thread is calling this routine.
async-cancel safe	The calling thread's cancelability type may be deferred or asynchronous when calling this routine.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the NOTES section below for more information on using this routine in a multi-threaded application.

PARAMETERS

name	Name for the log message (string, 6 bytes)

RETURN VALUE

Routine *zcomlname* has no return value.

NOTES

In a multi-threaded application, *zcomlname* is to be called only once by any one of the threads. It sets up the logging program name for the subsequent calls to *zcomlog* for all threads. If it is called again, the new program name is effective for the subsequent log messages.

EXAMPLE

```
#include   <zcom/zcomsys.h>
#include   <zcom/zcomcall.h>

char       name[6];

zcomlname  (name);
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zcomlog(3x).

ZCOMLOG (3X)

NAME

zcomlog – Add a message to the ZCOM log file

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

void zcomlog (error, msgno, narg [, desc, arg])

uint32  error;
int32   msgno;
uint8   narg;
uint32  desc;
uint32  arg;
```

DESCRIPTION

Routine *zcomlog* creates a log record and passes it to the ZCOM message log system, where the logged messages will be retrieved and processed by the ZMLOG program. Before calling this routine, *zcomlname*(3X) should be called to set up the logging program name for all subsequent log records.

The log record created consists of the program name as specified in the preceding *zcomlname* call, the message number *msgno* and the associated arguments *arg*. This routine has a variable number of arguments: the call argument pair "*desc, arg*" repeats for the number of times specified by *narg*.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.

async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the NOTES section for more information on using this routine in a multi-threaded application.

PARAMETERS

error Error code to be logged (*unsigned int*). The error code is displayed as a 5-digit unsigned value (ranges from 00000 to 99999) in the error message log file.

msgno ID number of the message to be logged (*signed int*).

narg Number of arguments to be logged with the message (*positive int*).

desc Argument descriptor used to describe the "type" of the argument which follows. The values of this parameter may be one of the following:

ZCOM_AT_CHAR	- char
ZCOM_AT_UCHAR	- unsigned char
ZCOM_AT_SINT	- short int
ZCOM_AT_USINT	- unsigned short int
ZCOM_AT_INT	- int
ZCOM_AT_STR	- char *
ZCOM_AT_ERRNO	- int (UNIX error code, usually errno) UNIX error string will be logged
ZCOM_AT_ZERROR	- int (ZCOM error code) ZCOM error string will be logged
ZCOM_AT_ZSTAT	- int (ZCOM status code) ZCOM status string will be logged
ZCOM_AT_XSTAT	- int (ZCOM X.25 status code) ZCOM X.25 status string will be logged

arg Argument value to insert into the message to be logged. Note that the argument's type must match the type specified in the *desc* parameter. Each argument will be substituted into the message text string using *printf*.

RETURN VALUE

Routine *zcomlog* has no return value.

NOTES

1. The ZMLOG program retrieves the unique message test using the logging program name and message number. By default, the message tests of all log messages are retrieved from the message file *default.msg* in */opt/acc/msg*. If the environment variable "LANG" is defined, the message text is retrieved from the message file *$LANG.msg* instead. In case it does not exist, the *default.msg* file is used. This allows support for multiple message file in different languages.
2. Before calling this routine, *zcomlname*(3X) should be called to set up the logging program name for all subsequent log records.

3. In a multi-threaded application, *zcomlname* is to be called only once by any one of the threads. It sets up the logging program name for the subsequent calls to *zcomlog* for all threads. If it is called again, the new program name is effective for the subsequent log messages.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

uint32    error;
int32     msgno;
uint32    desc;
uint32    arg;
char      p1;
int       p2;

error    = 1730;
msgno    = 150; /* Msg text = "This is an error. P1 = %c P2 = %d".
*/
p1       = 'a';
p2       = 37;
zcomlog (error, msgno, 2, ZCOM_AT_CHAR, p1, ZCOM_AT_INT, p2);
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).
/opt/acc/msg/default.msg	Master message text file.

SEE ALSO

zcomlname(3x), zmlog(1)

ZCOMSTATUS (3X)

NAME

zcomstatus – Return a formatted ZCOM status or error message

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

char   *zcomstatus (req, stat)

int32  req;
int32  stat;
```

DESCRIPTION

Routine *zcomstatus* returns a ZCOM status message string corresponding to a specific status code. The return character string has a length of 31 bytes maximum, plus a terminating zero. See *Multiprotocol ACC Programmers Reference Guide* under 'ZCOM Status Message Routine' in Appendix B, *ZCOM Status Codes*.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

not a cancellation point	Thread cancellation will not occur when a thread is calling this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

req	Request code (from the *zread* header, *zmhd*)
stat	Status (ZCOM error code if < 0)

NOTES

This routine reads a message file for the status message text. It uses the file *zstatmsg.msg* in /opt/acc/msg (ZCOM_MESSAGE_PATH). If not found, it uses the default message file *default.msg*.

RETURN VALUE

Routine *zcomstatus* returns a message string, corresponding to either the ZCOM status if *stat* is non-negative, or the ZCOM error if *stat* is negative.

EXAMPLE

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h>

char      *statmsg;  /* Returned message string */
int32     req;
int32     stat;

statmsg = zcomstatus (req, stat);
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).
/opt/acc/msg/zstatmsg.msg	ZCOM status messages.
/opt/acc/msg/default.msg	Master message text file.

ZCONFIG (3X)

NAME zconfig – ZCOM Dynamic System Configuration (DSC)

SYNOPSIS

```
#include   <zcom/zcomsys.h>
#include   <zcom/zcomcall.h>     /* If compliled with ANSI C (recommended) */

int32 zconfig (zap, mode, node, cdata, rdata)

zaddr_type *zap;
uint32 mode;
uint32 node;
zconfig_type *cdata;
zcfgret_type *rdata;
```

DESCRIPTION

Routine *zconfig* provides a general purpose mechanism for configuring a ZCOM system dynamically, i.e. altering some of the system tables or hardware settings while the system is running.

ATTENTION: The current release provides very limited user support for the functionality documented here. Specifically, the only supported functions are the Dynamic System Configuration (DSC) Control request to send a "Reconfiguration Complete" system event message and all of the Port Configuration actions.

The configuration modified by use of the *zconfig* routine is only effective while this instance of the ZCOM subsystem is running. The new configuration remains in force until changed by another call or until the ZCOM system is reinitialized. If the ZCOM subsystem is shutdown and restarted, the original configuration defined in the TTGEN configuration file is restored and all dynamic configuration changes are lost. Therefore, all permanent configuration information should be defined in the TTGEN configuration file.

This routine is intended for use by advanced user programs and should be used with care. Since this routine is used to dynamically modify the internal data structures used in the ZCOM subsystem, applications which use this functionality may need to be recompiled with new releases of the ACC base product.

The libraries *libzcom_c.a* and *libpthread.a* must be linked into the calling program by giving the options "*-lzcom_c -lpthread*" to cc(1) or ld(1)

Threads Considerations

This routine may be called from multi-threaded application using the POSIX (1003.1c) kernel threads API package. It has the following characteristics when called by multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork* (2) but before *exec* (2).

PARAMETERS

zap Pointer to source ZCOM address.

It is the ZCOM address identifying the originator of the configuration request and is usually set to the primary ZLU of the calling program. In this implementation, the originator must be from the local node, i.e. the "node" field of this address must be either zero or the local node number (except for the Port Configuration functions)

mode Operating mode.

The mode parameter specifies whether to perform the configuration with or with-out wait. Allowed values are:

ZcDSC_IMMEDIATE(0) - Immediate configuration.

RS 4 The caller is suspended until the requested configuration is completed. The *rdata* parameter contains the returned response data.

ZcDSC_DEFERRED (1) - Deferred configuration.

RS 4 ZCONFIG initiates the configuration and returns. On completion, a response message is sent to the "saddr". **This mode is not supported.**

node Destination node.

This is the node number of the remote (or local) ZCOM system where the configuration is to take place. If zero is specified, it is the local system. Note that the *node* parameter must specify the local system with this release (except for the Port Configuration functions)

cdata Configuration data.

This points to a data structure containing all the information necessary for the configuration. See below for a description of this structure.

rdata
(Return Param) Response data.

This points to a data structure containing the returned configuration response data. It is returned with valid data only when it is an immediate configuration (*mode* ZcDSC_IMMEDIATE) and *zconfig* returns with zero (i.e. no error). See the following for a description of this structure.

Configuration Data The configuration data structure consists of a *header* and a "union" *data* portion for configuration information

```
/* Configuration Header */
typedef struct {
    uint16       config;        /* Type of configuration */
    uint16       action;        /* Configuration action */
    uint16       tag1;          /* Tag parameter 1 */
    uint16       tag2;          /* Tag parameter 2 */
 } zcfghdr_type;

/* Configuration Data */
 typedef struct {
     zcfghdr_type        header;      /* Header Information */
     union {
         struct {
             uint32      cfg_class;   /* Class of Reconfig (1-999 Resvd) */
             uint32      cfg_info1;   /* Caller supplied information. */
         } setctl;                    /* Request = DSC Control */
         struct {
             char        sysname[80]; /* New ZCOM system name */
             uint32      qlimit;      /* Queue limit */
         } setsys;
         struct {
             uint16      iftno;       /* Interface card number */
             uint8       bc1_addr;    /* 1st level bus converter addr */
             uint8       bc_addr;     /* Bus converter addr of hardware */
```

```
            uint16      card_addr;   /* Card address of new hardware */
            uint16      itype;       /* Interface card type */
            char        fname[ZCOM_MAXFNAME];  /* Firmware filename */
        } setcard;                  /* Request=Interface Configuration */
        struct {
            uint16      iftno;       /* Interface card number */
            uint8       bc1_addr;    /* 1st level bus converter addr */
            uint16      bc_addr;     /* Bus converter addr of hardware */
            uint16      card_addr;   /* Card address of new hardware */
            uint16      itype;       /* Interface card type */
            zcfg_lookup_t cfglt;     /* Configuration lookup table info */
            char        fname[ZCOM_MAXFNAME];  /* Firmware filename */
        } setcard_mmp;               /* Request=Interface Configuration */
                                     /* For Multi-msg BP protocol. */
        struct {
            uint16      iftno;       /* Interface card number */
            uint16      portno;      /* Port number */
            uint16      rcode;       /* Request code */
            uint16      spare;       /* Reserved, not used */
            zpconf_type cnfg;        /* Port configuration word */
            uint32      qlimit;      /* Port tx limit */
        } setport;                   /* Request = Port Configuration */
        struct {
            zaddr_type  zaddr;       /* Terminal ZLU */
            zaddr_type  daddr;       /* Destination terminal ZLU */
            uint16      iftno;       /* Interface card number */
            uint16      portno;      /* Port number */
            uint16      subchno;     /* Subchannel number */
            uint16      dtype;       /* Terminal device type */
            uint16      ltype;       /* Logical terminal type */
            zltt_type   ltt;         /* Logical terminal data */
            zptt_type   ptt;         /* Physical terminal data */
        } setterm;                   /* Request=Terminal Configuration */
        struct {
            uint16      iftno;       /* Interface card number */
            uint16      portno;      /* Port number */
            uint16      subchno;     /* 1/4 Subchannel number */
            uint16      rcode;       /* Request code */
            zptcfg_t    *ptcfg;      /* Ptr to port & subch config data */
            uint32      len;         /* Lgth (bytes) of ptcfg->ptinfo */
            unit32      qlimit;      /* Port transmit limit */
        } setportsc;                 /* Request = Port Sub-Chan Config. */
    } data;
} zconfig_type;
```

The *header* specifies the configuration. It is returned without change in the response data, and may be used by the caller to identify a configuration request.

Field *header.config* specifies the type of configuration. The interpretation of the *header.action* and *data* fields vary depending on the configuration type. Allowed values are:

ZCOM_ZCONFIG_CNTL	- DSC Control
ZCOM_ZCONFIG_SYS	- System Configuration
ZCOM_ZCONFIG_CARD	- Interface Configuration
ZCOM_ZCONFIG_PORT	- Port Configuration
ZCOM_ZCONFIG_TERM	- Terminal Configuration
ZCOM_ZCONFIG_PORTSC	- Port and Subchannel Configuration

Field *header.action* specifies what action is to be taken for the specified configuration type. Its meaning depends on *header.config*.

Fields *header.tag1* and *header.tag2* are tag parameters and will be returned without change in the response data.

The union field *data* contains information for the configuration request. The relevant sub-field to be used depends on *header.config*.

Refer to the individual configuration sections for more information on configuration data.

Response Data

The response data structure consists of the header fields and a “union” *data* portion for completion response:

```
/* Configuration Response */
 typedef struct {
     zcfghdr_type        header;   /* Header Information */
     union {
         struct {
             int32       error;    /* Return ZCOM error */
         } ctlret;         struct {
             int32       error;    /* Return ZCOM error */

         } sysret;
         struct {
             uint32      iftno;    /* Interface card number */
             int32       error;    /* Return ZCOM error */
         } cardret;
         struct {
             uint32      iftno;    /* Interface card number */
             uint32      portno;   /* Port number */
             int32       status;   /* Return status */
```

```
        } portret;
        struct {
            zaddr_type  zaddr;    /* Terminal ZLU */
            int32       error;    /* Return ZCOM error */
        } termret;
        struct {
            uint32      iftno;    /* Interface card number */
            uint32      portno;   /* Port number */
            uint32      subchno;  /* Subchannel number */
            int32       status;   /* Return status */
        } portscret;
    } data;
} zcfgret_type;
```

The *header* is the same as in the configuration data. Its fields are returned without change for the caller to identify the response.

The union field *data* contains response data returned from the specified configuration. The relevant sub-field to be used depends on *header.config*.

Refer to the individual configuration sections for more information on response data.

TYPES OF CONFIGURATION FUNCTION

DSC Control

The DSC control function is to provide global control and status on the access of DSC functions. There are 3 DSC control actions.

Action ZcENABLE_DSC(1) is to enable the DSC access after it has been disabled. By default, the DSC access is enabled when ZCOM system is started up.

Action ZcDISABLE_DSC(2) is to disable DSC access. When disabled, local processes attempting DSC functions will be suspended. For DSC requests from remote system, ZCOM error -41 (ZEDSCLOCK, DSC function is disabled) is returned.

Action ZcDSC_SEND_RECFG(3) is used to broadcast a “Reconfiguration Complete” system event message to all interested programs and protocol drivers. This action should only be used after completing all zconfig() calls to dynamically reconfigure the ZCOM subsystem. Any process or protocol driver which has previously called the ZCOM zevent_rcvr() routine will be notified with a system event message. A dynamic reconfiguration complete event message returns an event type of ZCOM_SETYPE_RECONFIG.

Configuration Data

header.config

ZCOM_ZCONFIG_CNTL (DSC Control)

header.action

ZcENABLE_DSC	(1) - Enable DSC access
ZcDISABLE_DSC	(2) - Disable DSC access
ZcDSC_SEND_RECFG	(3) - Broadcast reconfiguration complete system event message.

header.tag1,
header.tag2

Any values.

data.setctl.cfg_class

Caller supplied type of reconfiguration. There is one predefined configuration class, *ZcCLASS_X25*. This event should be generated whenever an X.25 link is dynamically configured. This includes modifying the number and/or type of Virtual Circuits and creating or deleting an X.25 level 2 (LAPB) ZLU.

If the ZCOM Subsystem is used with X25/9000 and x25init is executed, this event will be generated if the configuration for an X.25 link is modified. This field is used for *action* 3 only.

data.setctl.cfg_info1

Caller supplied information that should be specific to the type of reconfiguration which took place. In the case of ZcCLASS_X25, this field must contain the X.25 Level 2 ZLU of the link's configuration that was modified. Note that even if the ZLUs for a link were deleted from the configuration, the old Level 2 ZLU must still be supplied. A positive ZLU number indicates the link has been added, while a negative ZLU value indicates the link was deleted. This field is used for *action* 3 only.

Response Data

No specific response data or error.

System Configuration

There are 5 system configuration actions. They are for altering parameters related to the whole ZCOM runtime subsystem. *Action* 1 is for changing the ZCOM runtime system name. The system name is for information purposes only. This function does not affect the operation of the ZCOM subsystem.

Action 2-5 modify various default queue limits in the ZCOM subsystem. Such default limits only affect newly created queues. To modify limits on existing queues, use the *zsetql*(3X) routine instead.

Configuration Data

header.config

ZCOM_ZCONFIG_SYS (system configuration)

header.action

ZcDSC_MOD_SYS_NAME	(1) - Modify ZCOM runtime system name.

ZcDSC_SET_PROG_QLIMIT	(2) - Set default program queue limit.
ZcDSC_SET_TERM_XMIT_QLIMIT	(3) - Set default terminal transmit queue limit.
ZcDSC_SET_TERM_UNACK_QLIMIT	(4) - Set default terminal "unack" queue limit.
ZcDSC_SET_PORT_XMIT_LIMIT	(5) - Set default port transmit limit.

header.tag1,
header.tag2

Any values.

data.setsys.sysname

New runtime ZCOM system name. This field is used for *action* 1 only.

data.setsys.qlimit

New default queue limit. This field is used for a*ction* 2 to 5 only. For program and terminal transmit queues, the limit is in number of messages. For the terminal "unack" queue and the port transmit limit, the limit is in bytes.

Response Data

data.sysret.error

This is the returned system configuration error:
0 - No error (successful)
!=0 - ZCOM error code

The error message text may be fetched using z*comerror*(3X) with a non-zero error code.

Interface Configuration

There are 6 interface configuration actions. They are used to alter the parameters related to an interface card, and for controlling the operation of an interface card.

Action ZcDSC_CHNG_FW_FILE (1) modifies the firmware file name (used to download the MUX card). This change will have no effect until the interface card is restarted, causing the new firmware file to be used to download the card.

Action ZcDSC_CREATE_IFT (2) allocates the internal data structures required for a new interface card. That is, it creates a new Interface Table entry.

Action ZcDSC_START_CARD (3) starts up an interface. This initializes the hardware, the ports and terminals according to the configuration kept in the ZCOM tables. In functionality, it is identical to a *zmon restart* command.

Action ZcDSC_HALT_CARD (4) halts the hardware and stops new requests to the ACC interface card. All pending buffers and terminals linked to the interface remain unchanged. This function prepares an Interface Table entry for reassignment to different hardware (a new MUX interface card).

Action ZcDSC_STOP_CARD (5) stops an interface card as in *action* ZcDSC_HALT_CARD. In addition, all buffers linked to the interface and its terminals are released. Terminals are then set to unavailable. This function may be used to prepares an interface for re-linking to different hardware, or to eliminate all pending resources for a failed card. It is identical to a *zmon shutdown* command.

Action ZcDSC_REASSIGN_IFT (6) associates an interface table entry with a different MUX interface card. An interface must be halted or stopped (i.e. *action* ZcDSC_HALT_CARD or ZcDSC_STOP_CARD) before this request can be issued.

Configuration Data

header.config

ZCOM_ZCONFIG_CARD (interface configuration)

header.action

ZcDSC_CHNG_FW_FILE	(1) - Modify interface firmware file name
ZcDSC_CREATE_IFT	(2) - Create a new interface table
ZcDSC_START_CARD	(3) - Startup (e.g. restart) an interface card
ZcDSC_HALT_CARD	(4) - Disable an interface
ZcDSC_STOP_CARD	(5) - Shutdown (e.g. disable and flush) an interface card
DSC_REASSIGN_IFT	(6) - Link an interface to different hardware

header.tag1,
header.tag2

Any values

data.setcard.iftno

Interface card number (0, 1,...)

data.setcard.bc1_addr

data.setcard.bc_addr

data.setcard.card_addr

These are the bus converter address and card address (as specified in a TTGEN configuration file) of the new hardware. These fields are needed for *action* 2 and 6 only.

data.setcard.itype

Interface card type. This indicates the hardware type of the new interface card. Currently, the types *ZCOM_ITYPE_MUX*, *ZCOM_ITYPE_Z7200A*, *ZCOM_ITYPE_Z7300A*, *ZCOM_ITYPE_Z7340A*, *ZCOM_ITYPE_Z7350A*, and *ZCOM_ITTYPE_Z7400A*R are supported. This field is needed for *action* 2 only.

data.setcard.fname

New firmware file name. This field is used for *action* 1 only.

Response Data

data.cardret.iftno

This is the interface card number as in the configuration data. It indicates the interface table entry that was created or modified.

data.cardret.error

This is the returned interface configuration error:
0 - No error (successful)
< 0 - ZCOM error code
The error message text may be fetched using \fIzcomerror\fR(3X) with a non-zero error code.

Port Configuration

There are 8 port configuration actions. *Actions* ZcDSC_ALL_PARMS, ZcDSC_CFG_MODE, ZcDSC_BAUD_RATE, and ZcDSC_PORT_MODE configure the datacomm ports on the MUX interface card. All terminal ZLUs on the port being changed must be **disabled** before the port configuration takes place (use *zcntl* to disable a terminal ZLU using request code ZCOM_MRQCODE_DSB[8]). Examine the TTGEN configuration file to determine which ports are defined for which MUX interface cards and the terminals associated with each port.

Action ZcDSC_BREAK_REQ is used to activate the Port Break Routine of the protocol firmware. Since Port configuration by-passes the normal terminal transmit queues, this function is used as a software interrupt to the protocol specific firmware to initiate a recovery process when there is a protocol problem and the transmit queues are “blocked up”. If the protocol for the specified port does not support this facility, error status 8 is returned.

Action ZcDSC_SET_XMIT_LIMIT changes the port transmit limit of a specific port on a MUX card. This limit controls the flow of requests from the host to the card for terminals on the related port.

Action ZcDSC_ENABLE_PORT enables a port after it has been disabled (*action* ZcDSC_DISABLE_PORT).

Action ZcDSC_DISABLE_PORT disables a port putting it into an inoperative state. All linked terminals are marked without actually sending a disable request to the interface card. This is to avoid a “hung” terminal disable request due to a hardware failure on a port. This action is designed to isolate a port that has failed from the ZCOM system. The ZCOM system will stop using the port and the terminals on the failed port may then be moved or deleted.

Configuration Data

header.config

ZCOM_ZCONFIG_PORT (port configuration)

header.action

ZcDSC_ALL_PARMS	(1) - Set all operating modes (Configuration, Baud Rate and Port)
ZcDSC_CFG_MODE	(2) - Set Configuration mode only (Ecode, Sync, Mode, Parity)
ZcDSC_BAUD_RATE	(3) - Set Baud Rate mode only (X.Clk, S.Clk, Baud Rate)
ZcDSC_PORT_MODE	(4) - Set Port mode only (Pmode)
ZcDSC_BREAK_REQ	(5) - Port Break request
ZcDSC_SET_XMIT_LIMIT	(6) - Modify port transmit limit
ZcDSC_ENABLE_PORT	(7) - Enable a port
ZcDSC_DISABLE_PORT	(8) - Disable a port

header.tag1,
header.tag2

Any values.

data.setport.iftno

Interface card number (0, 1,...).

data.setport.portno

Port number in the interface card:
0..7 For Z7200A, Z7340A, and Z7400A ACC interface cards
0..3 For the Z7300A ACC interface card
0..1 For the Z7350A ACC interface card

data.setport.rcode

This field is for *actions* 1-4 only. Values are:
1 - Use configuration in interface table
2 - Use data in field \fI*data.setport.cnfg*

data.setport.spare

Unused spare field. For data alignment.

data.setport.cnfg

Configuration data (32 bits). This field is used for *actions* 1 to 4 only. The content is dependent on the specific hardware being configured. It contains the following information:
For the Z7200A, Z7400A, Z7340A, Z7350A cards:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Ecode		Sync		Mode		Parity		X. Clk		S. Clk		Baud 1			
Pmode		Pmode2		Baud 2				0 = Not Used							

Ecode	*Value*	*Encoding modes*
	0 0	NRZ
	0 1	NRZI
	1 0	FM1 (transition = 1)
	1 1	FM0 (transition = 0)

Sync	*Value*	*Sync mode select*
	0 0	8-Bit sync (monosync)
	0 1	16-Bit sync (bisync)
	1 0	SDLC Mode
	1 1	External sync

Mode	*Value*	*Operating mode*
	0 0	Sync mode
	0 1	Async 1 stop
	1 0	Async 1.5 stop bits
	1 1	Async 2 stop bits

Parity	*Value*	*Parity select*
	0 0	No Parity
	0 1	Odd parity
	1 0	No Parity
	1 1	Even parity

X.Clk	*Value*	*Clock multiplier*
	0 0	X 1
	0 1	X 16
	1 0	X 32
	1 1	X 64

S.Clk	*Value*	*Clock source*
	0 0	External clock
	0 1	Internal clock from Baud Rate Generator (BRG)
	1 0	X.21 clock source
	1 1	DPPL output (must use BRG as source)

The baud rate is split between two 4-bit parameters (for compatibility reasons). If the Baud 1 parameter is 0, then Baud 2 is used. Rates listed in **bold** are not available on the Z7340A and Z7350A interface cards. Baud rates above 76800 are not supported on the Z7200A card and rates above 128000 are not supported for the Z7400A card.

Baud 1	*Value*	*Rate*	*Value*	*Rate*
	0000	Use Baud2	1000	19,200

	0001	**300**	1001	38,400
	0010	600	1010	48,000
	0011	1,200	1011	57,600
	0100	2,400	1100	76,800
	0101	4,800	1101	64,000
	0110	9,600	1110	128,000
	0111	14,400	1111	256,000

Baud 2	*Value*	*Rate*	*Value*	*Rate*
	0000	**150**	1000	768,000
	0001	56,000	1001	1,024,000
	0010	153,600	1010	1,228,800
	0011	307,200	1011	1,536,000
	0100	192,000	1100	1,544,000
	0101	614,400	1101	2,048,000
	0110	384,000	1110	Reserved
	0111	512,000	1111	Reserved

	Pmode	**Pmode2**	
Pmode	*Value*	*Value*	*Port mode*
	0 0	0 0	RS232/V.24 Mode
	0 1	0 0	RS422 Mode
	1 0	0 0	Loopback Mode (Tristate)
	1 1	0 0	X.21/V.11 Mode

0 0	0 1	V.35 (Z7340A and Z7350A only)
0 0	1 0	RS-449 (Z7340A and Z7350A only)
0 0	1 1	V.36 (Z7340A and Z7350A only)
All other values		Reserved

For the Z7300A (E1/T1) card:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Ecode		FMode				0		Fsync		S. Clk		0		TxAtt	
Pmode				0				QD	CR	NP	0	LH	CF	JP	0

Ecode *Value* *Encoding modes*

0 0	HDB3- High Density Bipolar/3 zeros (E1)
0 1	B8ZS- Bipolar/8 Zero Substitution (T1)
1 0	AMI- (Not Supported)
1 1	Not defined

Fmode Value *Frame mode parameter*

0000	T1: F4 - 4-frame Multiframe
0001	T1: SF - 12-frame Multiframe
0010	T1: ESF - Extended Superframe
0011	T1: F72 - Remote Switch mode
0100	E1: DF - Doubleframe
0101	E1: MF - CRC Multiframe

	0110	Transparent (voice) mode

Fsync	*Value*	*Other port for clock synchronization*
	0 0	Port 0
	0 1	Port 1
	1 0	Port 2
	1 1	Port 3

S.Clk	*Value*	*Clock source*
	0 0	External clock (slave)
	0 1	Internal clock (master)
	1 0	Clock from other port (specified in Fsync)
	1 1	Undefined

TxAtt	*Value*	*T1 Transmit Attenuation (Z7330B only)*
	0 0	0 dB
	0 1	-7.5 dB
	1 0	-15 dB
	1 1	-22.5 dB

Pmode	*Value*	*Port mode*
	0000	Reserved

...		
0100	Reserved	
0101	E1 twisted pair DB9	120 ohm
0110	E1 coax BNC	75 ohm
0111	T1 twisted pair RJ45	100 ohm
1000	Loopback Mode (Tristate)	
1001	E1 twisted pair RJ45	100 ohm
1010	T1 twisted pair DB9	120 ohm
1011	T1 coax BNC	75 ohm
1100	Reserved	
...		
1111	Reserved	

QD *1 = On* *Fast link-down option*

Makes the line go down whenever loss of T1/E1 frame synchronization is detected by the FALC. Otherwise by default there is a delay to avoid marking the line down on glitches.

CR *1 = On* *T1/ESF PRMs transmitted as "carrier"*

T1/ESF PRMs transmitted as “carrier”. The performance report messages transmitted in T1/ESF mode contain an address which indicates whether the ACC is a “network” or “user” device. By default the ACC is a “user” device.

NP *1 = On* *T1/ESF performance report messages suppressed*

This option suppresses the transmission of performance report messages in T1/ESF mode.

LH *1 = On* *Long-haul mode (Z7330B only)*

Enables the receive equalization and high power transmit line.

CF *1 = On* *Reduced corner frequency (Z7330B only)*

The corner frequency at which jitter attenuation begins, is reduced by a factor of ten by setting this option.

JP *1 = On* *JT G.706 mode (Z7330B only)*

In T1/ESF mode the CRC6 is calculated and the remote alarm processed according to JT G.706 (Japanese variation) by setting this option.

data.setport.qlimit

New port limit (in bytes). This field is for *action* ZcDSC_SET_XMIT_LIMIT only.

Response Data

data.portret.iftno
data.portret.portno

These are the interface and port numbers as supplied in the original configuration data (request). They indicate the interface and port whose configuration has been modified.

data.portret.status

Completion status returned from port configuration:

	< 0 - Standard ZCOM error.
PT_OK	(0) - No error (successful).
PT_INV_PORT	(1) - Port number out of range.
PT_BAD_PARM	(2) - Bad parameter.
PT_NOT_DSBL	(3) - Some terminals on this port not disabled.
PT_BAD_BAUD	(4) - Bad baud rate.
PT_ILL_SCLK	(5) - Illegal clock source.
PT_ILL_PMODE	(6) - Illegal port mode.
PT_BAD_PORT	(7) - Port failed self test.
PT_NO_BREAK	(8) - BREAK routine not available.
PT_BAD_XCLK	(9) - Clock multiplier not compatible with configuration mode.
PT_NO_PLL	(10) - No DPLL available at this baud rate.
PT_TOO_FAST	(11) - Async mode not valid for baud > 38400.

The error or status message text may be fetched using *zcomstatus* (3X) with *req* = ZCOM_MRQCODE_PORT (or 14) and the return status. Note that status codes PT_BAD_BAUD, PT_ILL_SCLK, PT_ILL_PMODE, and PT_BAD_PORT imply the associated port is now inoperative. Another port configuration request **must** be performed to recover it.

Port Subchannel Configuration

The port subchannel configuration is only valid for the E1/T1 interface. There are 12 port configuration actions. *Actions* ZcDSC_ALL_PARMS, ZcDSC_CFG_MODE, ZcDSC_BAUD_RATE, ZcDSC_PORT_MODE, ZcDSC_SET_TIMESLOTS and ZcDSC_SET_SUBC_SPECS configure the datacomm ports and/or subchannels on the MUX E1/T1 interface card. For actions other than ZcDSC_SET_TIMESLOTS and ZcDSC_SET_SUBC_SPECS all terminal ZLUs on the port being changed must be **disabled** before the port configuration takes place (use *zcntl* to disable a terminal using request code ZCOM_MRQCODE_DSB[8]). For actions ZcDSC_SET_TIMESLOTS and ZcDSC_SET_SUBC_SPECS all terminal ZLUs on the subchannel(s) being changed must be **disabled** before the port configuration takes place. Note that, if a timeslot is being reconfigured to a new subchannel both the old and new subchannels must have no **enabled** terminals, for the reconfiguration to proceed. Examine the TTGEN configuration file or use zinfo() to determine which ports/subchannels are defined for which MUX interface cards and the terminal ZLUs associated with each port.

Action ZcDSC_BREAK_REQ is used to activate the Port Break Routine of the protocol firmware. Since Port configuration by-passes the normal terminal transmit queues, this function is used as a software interrupt to the protocol specific firmware to initiate a recovery process when there is a protocol problem and the transmit queues are “blocked up”. If the protocol for the specified port does not support this facility, error status PT_NO_BREAK (8) is returned.

Action ZcDSC_SET_XMIT_LIMIT changes the port transmit limit of a specific port on a MUX card. This limit controls the processing of transmit requests for terminals on the related port.

Action ZcDSC_ENABLE_PORT enables a port after it has been disabled (*action* ZcDSC_DISABLE_PORT).

Action ZcDSC_DISABLE_PORT disables a port putting it into an inoperative state. All linked terminals are marked **disabled** without actually sending a disable request to the interface card. This is to avoid a “hung” terminal disable request due to a hardware failure on a port. This action is designed to isolate a port that has failed from the ZCOM system. The ZCOM system will stop using the port and the terminals on the failed port may then be moved or deleted.

Action ZcDSC_ENABLE_SUBC enables a subchannel after it has been disabled (*action* ZcDSC_DISABLE_SUBC).

Action ZcDSC_DISABLE_SUBC disables a subchannel putting it into an inoperative state. All linked terminal ZLUs are marked *disabled* without actually sending a disable request to the interface card. This is to avoid a "hung" terminal disable request due to a hardware failure on a subchannel. This action is designed to isolate a subchannel that has failed from the ZCOM system. The ZCOM system will stop using the subchannel and the terminals on the failed subchannel may then be moved or deleted.

Action ZcDSC_SET_TIMESLOTS is used to modify timeslot assignments. Individual timeslots can be configured, without affecting the operation of the remaining timeslots on the same port.

Action ZcDSC_SET_SUBC_SPECS is used to modify subchannel specifications. Individual subchannels can be configured, without affecting the operation of the remaining subchannels on the same port.

Configuration Data

header.config

ZCOM_ZCONFIG_PORTSC (port and subchannel configuration)

header.action

ZcDSC_ALL_PARMS	(1) - Set all operating modes (Configuration, Baud Rate and Port)
ZcDSC_CFG_MODE	(2) - Set Configuration mode only (Ecode, Sync, Mode, Parity)
ZcDSC_BAUD_RATE	(3) - Set Baud Rate mode only (S.Clk)
ZcDSC_PORT_MODE	(4) - Set Port mode only (Pmode)
ZcDSC_BREAK_REQ	(5) - Port Break request
ZcDSC_SET_XMIT_LIMIT	(6) - Modify port transmit limit
ZcDSC_ENABLE_PORT	(7) - Enable a port
ZcDSC_DISABLE_PORT	(8) - Disable a port
ZcDSC_ENABLE_SUBC	(9) - Enable a subchannel
ZcDSC_DISABLE_SUBC	(10) - Disable a subchannel
ZcDSC_SET_TIMESLOTS	(11) - Set timeslot assignment
ZcDSC_SET_SUBC_SPECS	(12) - Set subchannel specification

header.tag1,
header.tag2

Any values.

data.setportsc.iftno

Interface card number (0, 1,...).

*data.setportsc.port*no

Port number in E1/T1 interface card (0...3).

data.setportsc.subchno

Subchannel number within the port (0...31).

data.setportsc.rcode

This field is used for *actions* ZcDSC_ALL_PARMS, ZcDSC_CFG_MODE, ZcDSC_BAUD_RATE, ZcDSC_PORT_MODE, ZcDSC_SET_TIMESLOTS and ZcDSC_SET_SUBC_SPECS only. Values are:
1 - Use configuration in interface table
2 - Use data in field *data.setportsc.ptcfg*

data.setportsc.ptcfg

Pointer to the port and subchannel configuration data of which the data structure *zptcfg_t* is defined in */opt/acc/include/zcom/zcomsys.h.*

data.setportsc.ptcfg->ptcnfg

Port configuration data (32 bits) which is used for *actions* ZcDSC_ALL_PARMS, ZcDSC_CFG_MODE, ZcDSC_BAUD_RATE and ZcDSC_PORT_MODE only. See the Port Configuration section for the format of the configuration data (data.setport.cnfg field).

data.setportsc.ptcfg->ptinfo.subchb.tmsl[ ]

Timeslot assignment array: used for *actions* ZcDSC_ALL_PARMS or ZcDSC_SET_TIMESLOTS only. Up to 32 timeslot assignments are allowed. Each element of the array is indexed by the timeslot number. That is, timeslot 0 is in *tmsl[0]*, timeslot 1 is in tmsl[1], etc. For *action* ZcDSC_ALL_PARMS, all timeslots are configured. For *action* ZcDSC_SET_TIMESLOTS only timeslot entries with the UPD bit set are configured. The timeslot definition for each element in the array is shown below:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>UPD</td><td>TTI</td><td colspan="5">Transmit subch number</td><td colspan="8">Transmit fill mask</td></tr>
<tr><td colspan="2">Set to 0</td><td>RTI</td><td colspan="5">Receive subch number</td><td colspan="8">Receive fill mask</td></tr>
</table>

UPD: The Update bit is used with the ZcDSC_SET_TIMESLOTS action to identify timeslots which are to be configured. Timeslots with the UPD bit not set, are unaffected by the ZcDSC_SET_TIMESLOTS action.

Detailed descriptions of the TTI, RTI, Transmit and Receive subchannel numbers, and the Transmit or Receive fill masks, can be found in the Siemens MUNICH32 data sheet in *Section 4.2.4 Time Slot Assignment*.

A TTI (Transmit Timeslot Inhibit) bit value of '1' causes the transmit timeslot to be placed into tristate. The data is not destroyed but sent in the next non-tristated timeslot allocated to this subchannel.

A RTI (Receive Timeslot Inhibit) bit value of '1' causes all data received in this timeslot to be ignored. The subchannel does not process inbound data for this timeslot.

The Rxcv and Txmit subch nmber fields associate the specified subchannel number with the Transmit and Receive timeslot, respectively. That is, tmsl[3] is used to specify which subchannel(s) will be using timeslot 3 for transmitting and receiving data. Note that they normally are, but are not required to be the same subchannel number for receiving and transmitting.

The fill mask is used to indicate which bits of the transmit and receive timeslot holds data.

data.setportsc.ptcfg->ptinfo.subchb.spec[ ]

Subchannel specification data: used for *actions* ZcDSC_ALL_PARMS or ZcDSC_SET_SUBC_SPECS only. Up to 32 subchannel specifications are allowed. For *action* ZcDSC_ALL_PARMS, all subchannels are configured. For *action* ZcDSC_SET_SUBC_SPECS only subchannel entries with the UPD bit set are configured. The subchannel specification definition is shown below:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	UPD	ITBS						Set to 0							
Set to 0							0	INV	Set to 0				Mode		0

UPD: The Update bit is used with the ZcDSC_SET_SUBC_SPECS action to identify subchannels which are to be configured. Subchannels with the UPD bit not set, are unaffected by the ZcDSC_SET_SUBC_SPECS action.

ITBS: Individual Transmit Buffer Size specifies the part of the on-chip transmit buffer allocated to the subchannel. The buffer is 64 bytes long and each timeslot assigned to the subchannel uses two bytes. For example, in the case of a subchannel that uses three timeslots (3 x 64Kbps = 192Kbps), the ITBS value should be set to six (6).

INV: Inversion; if this bit is set, all data of the subchannel transmitted or received by the MUNICH32 is inverted.

Mode: Defines the transmission mode:
00 - Transparent Mode A
01 - Transparent Mode B or Transparent Mode R
10 - V.110/X.30 Mode
11 - HDLC Mode

data.setportsc.len

The length in bytes of the subchannel buffer. For *actions* ZcDSC_ALL_PARMS, ZcDSC_SET_TIMESLOTS and ZcDSC_SET_SUBC_SPECS this length parameter should be set to the size of the *data.setportsc.ptcfg->ptinfo.subchb* structure (e.g. “sizeof(subchbuf_def)”). For all other actions this length is zero.

data.setportsc.qlimit

New port limit (in bytes). This field is for *action* ZcDSC_SET_XMIT_LIMIT only.

Response Data

data.portscret.iftno,
data.portscret.portno,
data.portscret.subchno

These are the interface, port number and subchannel numbers as supplied in the original configuration data (request). They indicate the interface, port and subchannel whose configuration has been modified.

data.portscret.status

Completion status returned from port configuration:

	< 0 - Standard ZCOM error PT_OK
PT_OK	(0) - No error (successful)
PT_INV_PORT	(1) - Port number out of range
PT_BAD_PARM	(2) - Bad parameter
PT_NOT_DSBL	(3) - Some terminals in this port not disabled
PT_BAD_BAUD	(4) - Bad baud rate
PT_ILL_SCLK	(5) - Illegal clock source
PT_ILL_PMODE	(6) - Illegal port mode
PT_BAD_PORT	(7) - Port failed self test
PT_NO_BREAK	(8) - BREAK routine not available
PT_BAD_XCLK	(9) - Clock multiplier not compatible with configuration mode
PT_NO_PLL	(10) - No DPLL available at this baud rate
PT_TOO_FAST	(11) - Async mode not valid for baud > 38400
PT_NO_BUF	(12) - Subchannel buffer required
PT_SC_ENB	(13) - Subchannel must be disabled
PT_BAD_TS	(14) - Timeslot is not permitted

The error or status message text may be fetched using *zcomstatus* (3X) with *req* = ZCOM_MRQCODE_PORT (or 14) and the return status. Note that status codes PT_BAD_BAUD, PT_ILL_SCLK, PT_ILL_PMODE, and PT_BAD_PORT imply the associated port is now inoperative. Another port configuration request **must** be performed to recover it.

Terminal Configuration

There are 5 terminal configuration actions. They are for altering various internal linkages and terminal tables in the ZCOM runtime system memory.

Action ZcDSC_CREATE_LTT_PTT (1) creates a simple terminal that has one Logical Terminal Table (LTT) and one Physical Terminal Table (PTT).

NOTE

For this new ZCOM terminal, zconfig() allocates data structures in the kernel for ZCOM drivers to access. However, related data structures for this terminal must also be allocated on the target ACC card to complete the task. To allocate the related firmware data structures, use zcntl() with the ZCOM_MRQCODE_TERM rcode value (see the zcntl(3X) man page).

Action ZcDSC_CREATE_LTT_ON_PTT (2) creates and links an LTT to an existing terminal. The newly created LTT becomes an inbound multiplexing terminal and bit ZCOM_LTFLAG_IMX of field “ltaddr” is set automatically.

Action ZcDSC_CLEAR_TERM (3) clears a terminal for deletion. It flushes data buffers and removes the receivers of a terminal. It is usually used to prepare a terminal for deletion. Note that a terminal must be disabled before it can be cleared.

Action ZcDSC_DELETE_TERM (4) removes a terminal and releases its table space to the free pool. If a terminal has multiple LTTs linked to it, only the LTT is removed, the PTT is untouched. Otherwise, both the LTT and PTT are removed. If the PTT is to be removed, it must be disabled and have no pending or active I/O requests. *Action* ZcDSC_CLEAR_TERM is designed to “clear” a terminal for deletion.

Action ZcDSC_MOVE_TERM (5) is to re-link a terminal to a different port, on the same or different interface cards. After moving the terminal entry, the terminal must be re-configured, enabled and activated (i.e. ZCNTL request 6, 7 and 9) before it can be re-used. A terminal must be disabled and have no active I/O requests before it can be moved.

Configuration Data

header.config

ZCOM_ZCONFIG_TERM (terminal configuration)

header.action

ZcDSC_CREATE_LTT_PTT	(1) - Create a new terminal (LTT and PTT)
ZcDSC_CREATE_LTT_ON_PTT	(2) - Create a new LTT on an existing PTT
ZcDSC_CLEAR_TERM	(3) - Clear connection to an existing terminal
ZcDSC_DELETE_TERM	(4) - Delete an existing terminal
ZcDSC_MOVE_TERM	(5) - Move an existing terminal to another port

header.tag1,
header.tag2

Any values.

data.setterm.zaddr

ZLU of the related terminal. The "node" field is not used. For *action* 1 and 2, if ZLU is non-zero, it must be an unassigned one and ZLU checksum is not necessary. If ZLU is zero, the system chooses an arbitrary (unused) one. For *actions* ZcDSC_CLEAR_TERM, ZcDSC_DELETE_TERM, and ZcDSC_MOVE_TERM, the ZLU must be defined and the ZLU checksum must be valid.

data.setterm.daddr

ZLU of the destination terminal. It specifies the destination PTT to which the newly created terminal is linked. It must be a defined terminal and the checksum must be valid. The "node" field is not used. It is used for *action* ZcDSC_CREATE_LTT_ON_PTT only.

data.setterm.iftno

Interface card number (0, 1,...). It specifies to which interface card the new terminal is linked. This field is needed for *actions* ZcDSC_CREATE_LTT_PTT and ZcDSC_MOVE_TERM only.

data.setterm.portno

Port number in the interface card (0 to 7). It specifies to which port the new terminal is linked. This field is needed for *actions* ZcDSC_CREATE_LTT_PTT and ZcDSC_MOVE_TERM only.

data.setterm.subchno

Subchannel number on the port (0 to 31). It specifies the subchannel number the terminal will communicate over. Currently, only the Z7300A card supports subchannels. All other ACC interface card types must set this parameter to zero. This field is needed for *actions* ZcDSC_CREATE_LTT_PTT and ZcDSC_MOVE_TERM only.

data.setterm.dtype

Terminal device type. It is set up in the newly created terminal. This field is needed for *actions* ZcDSC_CREATE_LTT_PTT and ZcDSC_CREATE_LTT_ON_PTT only.

data.setterm.ltype

Logical terminal type. It is set up in the newly created terminal. This field is needed for *actions* ZcDSC_CREATE_LTT_PTT and ZcDSC_CREATE_LTT_ON_PTT only.

data.setterm.ltt

Logical terminal information. This contains the logical terminal information of the newly created terminal. Only the user maintainable fields (i.e. “ltaddr” and the following fields) are copied to the new logical terminal table. The other fields are ignored. This field is needed for *actions* ZcDSC_CREATE_LTT_PTT and ZcDSC_CREATE_LTT_ON_PTT only.

data.setterm.ptt

Physical terminal information. This contains the physical terminal information of the newly created terminal. Only the user maintainable fields (i.e. “ptpoll” and the following fields) are copied to the new physical terminal table. The other fields are ignored. In addition, the ptmx_term, subch, pendg_txreq, scfg_len, and spec_cfg fields are also ignored. This is used for *action* ZcDSC_CREATE_LTT_PTT only.

Response Data

data.termret.zaddr

This is the terminal ZLU which has been successfully created or modified. For *actions* ZcDSC_CREATE_LTT_PTT and ZcDSC_CREATE_LTT_ON_PTT, it is the newly created terminal ZLU with the correct checksum.

data.termret.error

This is the returned terminal configuration error:

0 - No error (successful)
< 0 - ZCOM error code
> 0 - zcntl() firmware status code

NOTES

1. For port configuration, it is possible to change only one byte of the 32-bit configuration word, by using an appropriate value in *action*. In this case, the other bytes remain unaffected. Besides, *actions* 1-5 correspond to the functions supported by *zport*(3X), except that *zport* (3X) does not allow remote port configuration.
2. The following is the *zpconf_type* data structure definition (from <*zcom/zcomsys.h*>) used in port configuration. This structure allows the *cnfg* data to be referenced by individual bit fields (*zpconf_type.bits.<field>*), by individual configuration bytes (*zpconf_type.pconfig[<bytenum>*]), or by its entire 32-bit value (*zpconf_type.pcval*).

```
/*-- Port Configuration format --*/
 typedef union {
     struct {
         unsigned int ecode   : 2;   /* Encoding modes */
         unsigned int sync    : 2;   /* Sync mode select */
         unsigned int mode    : 2;   /* Operating mode */
         unsigned int parity  : 2;   /* Parity select */
         unsigned int xclock  : 2;   /* Clock multiplier */
         unsigned int sclock  : 2;   /* Clock source */
         unsigned int baud    : 4;   /* Baud rate value */
         unsigned int pmode   : 2;   /* Port mode select - original */

         unsigned int pmode2  : 2;   /* Port mode select - extended */
         unsigned int baud2   : 4;   /* Extended baud rate value */
         unsigned int spare   : 8;   /* Reserved, must be 0 */
     } bits;
     struct {
         unsigned int ecode   : 2;   /* Encoding modes */
         unsigned int frmode  : 4;   /* Framing mode */
         unsigned int set0_1  : 4;   /* Reserved, must be 0 */
         unsigned int sclk    : 2;   /* Clock source */
```

```
        unsigned int set0_2  : 4;   /* Reserved, must be 0 */
        unsigned int pmode   : 4;   /* Port mode */
        unsigned int set0_3  : 12;  /* Reserved, must be 0 */
    } e1t1_bits;
    int32 pcval;        /* Must be signed, -1 means bad config */
    uint8 pconfig[4];   /* 4 configuration bytes */ } zpconf_type;
```

3. While clearing a terminal (Terminal Configuration *action* ZcDSC_CLEAR_TERM), the terminal unacknowledged transmit queue is also flushed. If the terminal has pending transactions to the interface card, it will cause write completion mismatch errors to be logged. It is not critical if caused by a terminal clear operation. The driver error messages complaining about write completion errors may be safely ignored.

RETURN VALUE

zconfig returns **0** if the configuration request is accepted. A **non-zero** ZCOM error is returned if there is a problem with the request. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

For immediate configuration requests, a zero return indicates the request is complete and *rdata* should be checked for any configuration problem. A non-zero return value means the request is rejected and no valid data is returned in *rdata*.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>
 int32           ierr;
 zaddr_type      zaddr;
 uint32          node;
 zconfig_type    cdata;
 zcfgret_type    rdata;

cdata.header.config = ZCOM_ZCONFIG_PORT; cdata.header.action = ZcDSC_ALL_PARMS;
cdata.header.tag1   = 1; cdata.header.tag2   = 2; cdata.data.setport.iftno  = 0;
cdata.data.setport.portno = 4; cdata.data.setport.rcode  = 2;
cdata.data.setport.cnfg.pcval = 0x07178000;

if (ierr = \fBzconfig\fR (&zaddr, 0, node, &cdata, &rdata)) {      /* error return code
*/ } else {          /* good return code */ }
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zsetql(3X), zcntl(3X).

ZEVENT_RCVR (3X)

NAME

zevent_rcvr – Set up a program ZLU to receive ZCOM alarms and events

SYNOPSIS

```
#include          <zcom/zcomsys.h>
#include          <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int zevent_rcvr (rzap, action, eclass, einfop)

zaddr_type        *rzap;
int               action;
uint32            eclass;
Zclinfo_type      *einfop;
```

DESCRIPTION

Routine *zevent_rcvr* is used to set up a program to receive ZCOM alarm and system event messages. When this routine is called, it causes the ZCOM subsystem to queue the indicated type(s) of events or alarms on the program queue specified by the *rzap* parameter. The event type is specified through the *eclass* parameter. The application program can read the event and/or alarm messages by issuing a *zread* call using the program ZLU specified in the *rzap* parameter. There is a limit on the number of programs that can request reception of each source of ZCOM event messages (e.g. status event of different target nodes are considered different sources). The limit is set to ZcMAX_SHARED_RCVRS, currently, 64.

If the *action* parameter is ZcDELETE, then the receiver linkage is deleted. That is, events and alarms of the indicated type (eclass) will no longer be delivered to that receiver ZLU.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.

async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

rzap ZCOM address of the receiver program (required).

action Type of action to perform (required).

ZcDELETE_EVENT - Remove the linkage for the indicated event classes. That is, those event classes will no longer be delivered to the program queue specified by the *rzap* parameter. Events that have already been queued prior to this call are not deleted.

ZcADD_EVENT - Incrementally add the event classes indicated by the *eclass* parameter to the list of events delivered to the program (specified by the *rzap* parameter).

ZcREPLACE_EVENT - Atomically replace any previously specified event classes with new event classes specified by the *eclass* and *einfop* parameters.

eclass ZCOM Subsystem event class (type of event). Note that each event class is a single bit that can be OR'd together to specify multiple event classes in a single *zevent_rcvr* call.

ZcNODE_STATUS - Deliver an event when a Node or any link to a Node changes status. See the ZCOM Programmer's reference manual for a detailed discussion of node status events.

ZcDYN_CONFIG - Deliver dynamic reconfiguration completion events. These types of events are generated by the *zconfig* routine after an application or protocol driver has dynamically changed the ZCOM configuration.

ZcALL_EVENTS - Deliver all classes of ZCOM alarms and events.

einfop This data structure contains any event class specific information needed to carry out the *action* specified. Some of this information is optional, some of it is required (depending on the action and eclass values). The fields in this data structure are listed below with their dependent class types given in parenthesis.

src_node (ZcNODE_STATUS) - This is the node that should be used to detect the node status changes. It can be the local or a remote node. This field is required when ZcNODE_STATUS is specified in the *eclass* parameter.

target_node (ZcNODE_STATUS) - This is the node that should be monitored. If the *src_node* detects a change in the status of the target_node or the link between the src_node and target_nodes, an event message will be generated. If *target_node* is assigned the value ZcALL_NODES, then all nodes (from the src_node's perspective) will be monitored. This field is required when ZcNODE_STATUS is specified in the *eclass* parameter.

NOTES

When an Application program issues a *zread* call, it can differentiate the type of information it is receiving by looking at the type of the message. ZCOM Event messages will return a message type of 5 (ZCOM_MSTYPE_LSEM) for a local system event, and a message type of 13 (ZCOM_MSTYPE_RSEM) for a remote system event. The difference between local and remote event messages is where the event was generated. Local event messages were generated locally and remote event messages were generated at a remote node. The content of the message will contain additional information about the event.

The 'C' structure for a ZCOM Event Message is shown below:

```
/*--- ZCOM Event Messsage Type ---*/

typedef struct

uint8     setype;            /* Event type. */
uint8     seflag;            /* Event flags (reserved for now). */
uint16    senode;            /* Node where event was generated. */
uint32    setime;            /* Event time (Epoc time). */
union
      {
      struct {                     /* Node Status change Event data. */
           uint16  node_num;       /* Node whose state has changed. */
           int8    nstat;          /* Node state change indicator. */
           int8    spare;          /* Reserved for future use. */
           int8    hstat[ZCOM_NHOST];   /* Link state change indicator */
           } node;
      struct {                     /* Dynamic Reconfiguration Event data. */
           uint32  cfg_class;           /* User supplied config class
                                   /*   (1-999 reserved for use by ZCOM) */

           uint32  cfg_info1;           /* User supplied class specific info. */
           } recfg;
      } sedata;                    /* Event class specific data. */
} zevent_type;
```

For the field *setype* there are two defined event types:

ZCOM_SETYPE_NODE	- Node Status Change Event.
ZCOM_SETYPE_RECONFIG	- Reconfiguration Complete Event.

See the ZCOM *Programmer's Reference Manual* for a detailed description of the fields in this data structure.

RETURN VALUE Routine *zevent_rcvr* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include <zcom/zcomsys.h>
#include <zcom/zcomcall.h>

int           ierr;
zaddr_type    rzap;
int           action;
uin32         eclass;
Zclinfo_type  cinfo;

if (ierr = zevent_rcvr (&rzap, action, eclass, &cinfo)) {
 /* error return code */
}
else {
          /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zread(3X), zconfig(3X).

ZGET_SHRCVR_LIST (3X)

NAME

zget_shrcvr_list – Get list of current shared receivers

SYNOPSIS

```
#include <zcom/zcomsys.h>
#include <zcom/zcomcall.h>   /* if compiled with ANSI C (recommended) */

int32 zget_shrcvr_list (zap, mode, mlen, zrcvrs)

zaddr_type     *zap;
uint32         mode;
int32          mlen;
zaddr_type      zrcvrs[];
```

DESCRIPTION

Routine *zget_shrcvr_list* is used to retrieve the contents of a linked list of shared receivers from the ZCOM subsystem tables. Depending on the mode selected, and terminal specified (if needed), this call will return an array of shared receivers from one of the ZCOM tables (DSC Events, Node Status Receivers, PTT Receivers, or LTT Receivers).

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.

fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2). See the NOTES section below for more information on using this routine in a multi-threaded application.

PARAMETERS

zap	ZCOM address pointer. The uses of *node*, *zlu* and *zcs* depend on the specified mode. But in general this is used to select the specific table entry from which you want the shared receivers list.
mode	Specifies which shared receiver list to return. 0 - Logical Terminal Table Shared Receivers (*zap->zlu* is the ZLU of a terminal entry in the LTT, *node* and *zcs* are not used) 1 - Physical Terminal Table Shared Receivers for “normal” data and Status messages (*zap->zlu* is the ZLU of an entry in the PTT). 2 - Physical Terminal Table Shared Receivers for “control” messages (*zap->zlu* is the ZLU of a terminal in the PTT). 3 - DSC Event Receivers (The *zap* parameter is not used) 4 - Node Status Event Receivers (*zap->node* is the required node number, *zcs* and *zlu* are not used)
mlen	Maximum number of receivers to be returned in the *zrcvrs* array. To get the full list of receivers, this should be ZcMAX_SHARED_RCVRS+1.
zrcvrs **(Return Param)**	This is a returned array of all shared receivers currently configured. Each entry in the array is a single receiver for the type of data specified by the *mode* parameter. The list is terminated by an entry with a zlu value of 0. The maximum number of receivers in each type of shared receiver list is set to ZcMAX_SHARED_RCVRS (currently, 64). Hence, to get the full list of receivers, this array should have ZcMAX_SHARED_RCVRS+1 entries.

RETURN VALUE

Routine *zget_shrcvrs_list* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include   <zcom/zcomsys.h>
#include   <zcom/zcomcall.h>

int32         ierr;
int32         mlen;
uint32        mode;
zaddr_type    zaddr;
zaddr_type    rcvrs[size]; /* "size" is user-determined, and is >=
mlen */

mlen = size;
mode = 3;  /* Get list of receivers waiting for Dynamic Reconfig
Events */
if (ierr = zget_shrcvr_list (&zaddr, mode, mlen, rcvrs)) {
          /* error return code */
}
else {
          /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zevent_rcvr(3X), zconfig(3X)

ZINFO (3X)

NAME

zinfo – Get ZCOM table information

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */
int32 zinfo (zap, mode, ibuf, len)
zaddr_type  *zap;
unit32      mode;
char        *ibuf;
int32       len;
```

DESCRIPTION

Routine *zinfo* is used to retrieve the contents of various ZCOM subsystem tables. Depending on the mode selected, it retrieves information such as the System Header, Interface Table, Logical and Physical Terminal Tables, ZLU descriptor table, or Node Entry table.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2). See the NOTES section below for more information on using this routine in a multi-threaded application.

PARAMETERS

zap — ZCOM address pointer. The *node* field is used to indicate from which node to retrieve the information. The uses of *zlu* and *zcs* depend on the specified mode.

mode — Type of information to return:

ZcZLU_TBL	(0) - ZLU Descriptor information
ZcLTT_TBL	(1) - Logical Terminal Table information
ZcPTT_TBL	(2) - Physical Terminal Table information
ZcHDR_TBL	(3) - Header information (*zlu* and *zcs* are not used)
ZcIFT_TBL	(4) - Interface Table (*zlu* is the interface number, *zcs* is not used)
ZcNODE_BYNUM	(5) - Node entry (*zlu* is the required node number, *zcs* is not used)
ZcNODE_BYIDX	(6) - Node entry (*zlu* is the index within the node table, ranging from 0 to maximum-1, *zcs* is not used)

Bit 15 - When this bit (**ZCOM_ZINFO_NAVL**) is set, *zinfo* will return the configured ZLU, Logical and Physical terminal information for non-available terminals. A non-available terminal ZLU would normally result in a “Terminal is not available” error (**ZENAVL[-74]**). This bit has no effect for modes ZcHDR_TBL, ZcIFT_TBL, ZcNODE_BYNUM or ZcNODE_BYIDX.

ibuf (Return Param) — Buffer for return data. The full description of the layout of the return buffer is contained in the Chapter 3, *Tables and Data Structures* section.

len — Maximum number of bytes to be returned in *ibuf*. If the specified length is smaller than the requested table size, a truncated section of the table is returned. If the specified length is larger, only the size of the requested table is returned.

NOTES

The returned logical terminal table (*mode* = ZcLTT_TBL) consists of the basic table and the logical terminal table extension (the extension size is configurable in the TTGEN configuration file). The size of this buffer should be large enough to hold the required information.

ATTENTION

Note that this call returns the contents of the internal tables used to implement the ZCOM subsystem. As such, these tables are subject to change from release-to-release and may require an application be recompiled in order to function correctly on a new release. The release notes for each new release will indicate whether a recomplilation is required.

RETURN VALUE

Routine *zinfo* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h>

int32        ierr;
zaddr_type   zaddr;
uint32       mode;
char         ibuf[size]; /* "size" is user-determined, and is >= len */
int32        len;

if (ierr = zinfo (&zaddr, mode, ibuf, len)) {
          /* error return code */
}
else {
          /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.

/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZINIT (3X)

NAME

zinit – ZCOM routine initialization

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zinit      (sip)
zsinfo_type       *sip;
```

DESCRIPTION

Routine *zinit* initializes the access to the ZCOM system and sets up the global variables for use by the other ZCOM routines. Therefore, it must be called before any other ZCOM routines are used. This routine also returns some ZCOM system information. Since not every application programs require this information, so after the *zinit* call the buffer may be reused for other purposes.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

not a cancellation point	Thread cancellation will not occur when a thread is calling this routine.
async-cancel safe	The calling thread's cancelability type may be deferred or asynchronous when calling this routine.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the NOTES section below for more information on using this routine in a multi-threaded application.

PARAMETERS *sip (Return Param)* ZCOM system information data structure containing the following fields:

sizrev ZCOM software revision code (4 digits).

sinode ZCOM system local node number.

sinzlu Total number of ZLU tables in the ZCOM system.

sinltt Total number of Logical Terminal Tables in the ZCOM system.

sinptt Total number of Physical Terminal Tables in the ZCOM system.

sinift Total number of Interface Tables in the ZCOM system.

sintzlu Total number of terminal ZLUs in the ZCOM system.

siszlu1 First program ZLU in the ZLU table.

sipqlim Default program ZLU queue limit.

siltsze Size (in bytes) of the Logical Terminal Table (LTT).

sispare Reserved for future use.

NOTES

1. If the calling program has already opened a primary ZLU, the primary ZLU will be reset after calling *zinit*: the ZLU remains open and its contents is not disturbed, but it stops being the 'primary' ZLU of the calling program. The calling program will no longer have a primary ZLU and need to call *zopen* to open a primary ZLU again (usually with the same name, hence the same ZLU will be opened). The other opened ZLUs are not affected by calling *zinit*. See *zopen*(3X) for more information about the primary ZLU.
2. The *zinit* routine will reset the node timeouts to the default values. Hence, any user specified node timeout must be set again after calling *zinit*. See *zrntimer*(3X) for more information.
3. In a multi-threaded application, this routine is to be called only once by any one of the threads. Once *zinit* returns successfully, the necessary initialization tasks are completed for all threads, and all subsequent calls to ZCOM APIs will be allowed. There is no need for the other running threads or the newly created threads to call this routine again. Usually, *zinit* is to be called during the main program's initialization phase, before any threads are generated.

4. In a multi-threaded application, if this routine is called the second time by a thread while some ZCOM APIs are in-progress in the other threads, the access to the ZCOM system will be re-initialized. The in-progress APIs will detect this and 'retry' internally. This is handled automatically and is transparent to the application.

RETURN VALUE

Routine *zinit* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h>

int32         ierr;
zsinfo_type   sip;

if (ierr = zinit (&sip)) {
          /* error return code */
}
else {
          /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zopen(3X), zrntimer(3X).

ZLTMG (3X)

NAME

zltmg – Move a logical terminal between groups

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zltmg (zap, nap)

zaddr_type     *zap;
zaddr_type     *nap;
```

DESCRIPTION

Routine *zltmg* moves the logical terminal associated with a terminal ZLU to a new group of logical terminals designated by another terminal ZLU. The first terminal must not be the only terminal linked in its existing group.

After a successful move, the moved terminal will be a member of the new group, and its MUX, PORT, and physical terminal linkage are modified to match the new group.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zaddr	Terminal to be moved
nzaddr	A group member of the new group

NOTES

In the ZCOM subsystem, a group of logical terminals may be defined to link to a single physical terminal. This routine allows a logical terminal to be moved from one terminal group to another, thus all its physical input/output may be handled by a different physical device.

This call is intended for advanced ZCOM programmers, and should be used with care.

RETURN VALUE

Routine *zltmg* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h>

int32         ierr;
zaddr_type    zaddr;
zaddr_type    nzaddr;

if (ierr = zltmg (&zaddr, &nzaddr)) {
        <error return code>
}
else {
        <good return code>
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. (**Note**: must be the first include file before any other ZCOM include files)
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (**Note**: requires ANSI C compilation)

ZLTMX (3X)

NAME

zltmx – Control logical terminal multiplexing

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zltmx (zap, mode)

zaddr_type   *zap;
int32        mode;
```

DESCRIPTION

Routine zltmx is used to enable or disable the multiplexing of logical terminals. This routine allows dynamic control of the ZCOM_LTFLAG_OMX and ZCOM_LTFLAG_IMX logical terminal flag bits. These flag bits are initialized for each terminal ZLU by TTGEN, and may be changed by use of this call.

After calling this routine, the subsequent zrcvr and zsend calls will work according to the new settings.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address.	
mode	Multiplexing control mode	
	ZcENB_OUTB_MLTPLXG	(1) - Enable outbound multiplexing
	ZcENB_INB_MLTPLXG	(2) - Enable inbound multiplexing
	ZcDSB_OUTB_MLTPLXG	(3) - Disable outbound multiplexing
	ZcDSB_INB_MLTPLXG	(4) - Disable inbound multiplexing

NOTES

For a more detailed description of the multiplexing facilities available under ZCOM, see the section of the manual under the heading “ZCOM Message Handling”.

RETURN VALUE

Routine *zltmx* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include          <zcom/zcomcall.h>

int32          ierr;
zaddr_type     zaddr;
int32          mode;

if (ierr = zltmx (&zaddr, mode)) {
       /* error return code */
}
else {
       /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZLTQUEUE (3X)

NAME

zltqueue – Logical data queue allocation

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compliled with ANSI C (recommended) */

int32      zltqueue (appln, label, queuep)

uint32     appln;
char       *label;
uint32     *queuep;
```

DESCRIPTION

Routine *zltqueue* requests a data queue in the logical terminal tables for terminals of a specific application number. There is currently a maximum of 4 queues. Each queue is identified by the terminal application group and a user-defined label. The returned queue number (on successful completion) could then be used in subsequent data storing and retrieval calls (*ltqdput* and *ltqdget*).

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

appln	Application number of terminal group.
label	Queue label name.
queuep **(Return Param)**	Data queue number (or ZCOM error, if negative)

NOTES

The data queue returned should only be used for terminals within the specified application group (there is NO queue ownership validation when it is used in *ltqdget* and *ltqdput*). An application value of **zero** means terminals in **ALL** application groups. This **zero** application value allows the queue to be used to store global data for all terminals in the system.

Routine *zltqueue* returns an already assigned queue if the queue label matches a previously assigned queue (in this case the specified application number is not used or validated). This provides a facility to share a data queue between different programs and enables program restartability.

The following example illustrates the queue allocation algorithm:

Appln Program	Appln Nmbr	Appln Label	Queue Queue No.	Returned
A	30	USAGE 1	1	<-- first allocation
B	0	GLOBAL	2	<-- global allocation
C	40	CCCCC C	1	<-- different appln group
D	30	USAGE 2	3	<-- same appln, diff usage
E	30	USAGE 1	1	<-- data queue sharing

Appln Program	Appln Nmbr	Appln Label	Queue Queue No.	Returned
F	30	USAGE 3	4	<-- same appln, diff usage
G	30	USAGE 4	error	<-- no more queues for apln 30

Queues 1, 3, and 4 are allocated for application 30, and queue 2 is allocated for global usage in program B. Thus, because there is a current limit of four queues, an error is returned to program G when it attempts to allocate another queue to application 30.

RETURN VALUE

Routine *zltqueue* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32          ierr;
uint32         appln;
char           label[6];
uint32         queue

if (ierr = zltqueue (appln, label, &queue)) {
       /* error return code */
}
else {
       /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

ltqdget(3X), ltqdput(3X), zltstore(3X), zltup(3X)

ZLTSTORE (3X)

NAME

zltstore – Logical data storage allocation

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zltstore (appln, label, size, offsetp

uint32   appln;
char     *label;
uint32   size;
int32    *offsetp;
```

DESCRIPTION

Routine *zltstore* requests a data area in the logical terminal table extension for terminals of a specific application number. Each data area is identified by the terminal application group and a user defined label. The returned offset (on successful completion) could then be used in subsequent data storing and retrieval calls (*zltup*). For data update and retrieval, using data storage is relatively more efficient than using a data queue. However, due to the limited usable area in the terminal tables, a data queue should be used for large amounts of data. See under *zltqueue* for how to allocate a data queue.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.

fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

appln	Application number.
label	Storage label name.
size	Storage size requested, in bytes. The size must be a multiple of 4 bytes and must not exceed the available area in logical terminal table extension (as defined by LOGICAL-SIZE in TTGEN configuration file). Otherwise, error ZEBADSIZE (-63) is returned.
offsetp **(Return Param)**	Starting byte offset, relative to the beginning logical terminal table, for the storage in the extension area.

NOTES

The returned offset indicates the start of the usable data area, with a size as specified in the call. The actual size of the reserved area will be silently rounded up to a multiple of 8-byte. This is to ensure all the storage areas are aligned on a 8-byte boundary, hence the storage may be used to store 64-bit entity (e.g. *long long* type in 32-bit mode, or *long* and pointer types in LP64 mode) without data alignment problem.

The data area should only be used for terminals within the specified application group. There is NO queue ownership validation when it is used in zltup. An application value of zero is used for terminals of **ALL** application groups. A **zero** application value is used to store global data for all terminals in the ZCOM subsystem.

Routine *zltstore* returns an already assigned area if the application number **and** storage label **and** storage size all match a previously assigned area. Otherwise, an error is returned. This provides a facility to share data areas between different programs and enables program restartability.

If bit ZCOM_ZLTSTORE_NA (0x8000) of the *size* parameter is set before *zltstore* is called, then *zltstore* will only return a previously assigned area. If a matching area of storage is not assigned, this will result in an error return. If this bit of the *size* parameter is **zero**, a new storage area will be allocated if it is not already assigned.

RETURN VALUE Routine *zltstore* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include <zcom/zcomsys.h>
#include <zcom/zcomcall.h>

int32         ierr;
uint32        appln;
char          label[6];
uint32        size;
int32         offset;

if (ierr = zltstore (appln, label, size, &offset)) {
       /* error return code */
}
else {
       /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

ltqdget(3X), ltqdput(3X), zltqueue(3X), zltup(3X)

ZLTUP (3X)

NAME

zltup – Update logical terminal table

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zltup (zap, ibuf, len, offset)

zaddr_type     *zap;
char           *ibuf;
int32          len;
int32          offset;
```

DESCRIPTION

Routine *zltup* is used to update information in the user area of the logical terminal area. This user area contains information such as details pertaining to the location of the terminal, a user-assigned terminal name and the extension area (of configurable size). For complete details see the section on logical terminal tables.

Part of the logical terminal table is assigned by TTGEN, the rest is initialized according to the parameters specified in the TTGEN configuration file. The table extension (if defined) is set up and initialized to zero by the system (unless LDATA is specified in the TTGEN configuration file to pre-define the values).

Typically, *zltup* is used to update the user data kept in the table extension area.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.

async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address.
ibuf	Data buffer.
len	Byte length of buffer.
offset	Byte offset within the logical terminal table where the data in the buffer should be placed. See NOTES for information on finding the offsets.

NOTES

The *offset* returned by *zltstore* may be passed to *zltup* to update the allocated storage area in the extension area. To update the fields in the basic table, the offset may be worked out by the ZCOM_OFFSET macro (defined in *zcomsys.h*). For example, to get the offset for the application number field *ltapno*:

```
int32         offset;
zltt_type     ltt, *lttp;

offset = ZCOM_OFFSET (*lttp, ltapno);   <-- Both will work
offset = ZCOM_OFFSET (ltt, ltapno);     <-- Both will work
```

RETURN VALUE

Routine *zltup* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32             ierr;
zaddr_type        zaddr;
char              ibuf[size];  /* where size is user-defined; must
be >= len */
```

```
int32             len;
int32              offset;

if (ierr = zltup (&zaddr, ibuf, len, offset)) {
       /* error return code */
}
else {
       /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

ltqdget(3X), ltqdput(3X), zltqueue(3X), zltstore(3X)

ZLUOPEN (3X)

NAME

zluopen – Open a ZLU file

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int zluopen (zap, mode, fd)

zaddr_type          *zap;
int                 mode;
int                 *fd;
```

DESCRIPTION

Routine *zluopen* opens the device file associated with a program ZLU for use with the standard HP-UX file access routines. The *zap* parameter points to the program ZLU to be accessed. If successful, a file descriptor for the opened program ZLU device file is returned through the *fd* pointer. The *mode* parameter is intended for a future enhancement and is not used at this time. For future compatibility, *mode* should be set to zero.

The *zluopen* routine only allows opening the device files for the range of program ZLUs in the local system. If *zap* specifies a terminal ZLU, or a ZLU on remote node, the request will also be rejected with an error.

In the current implementation of ZCOM, only the *select* and *close* functions are allowed on the returned file descriptor. Other operations (such as *read* and *write*) will result in error *ENOSYS* (Function is not available). A *select* for *read* on the returned file descriptor is to check whether input data is available on a program ZLU. A *select* on *write* on the returned file descriptor will always return TRUE, while a *select* on *exception* always returns FALSE.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address (checksum is not used). Node must be zero or contain the local node as defined in the TTGEN configuration file.
mode	Open mode. Parameter is not used and should be specified as zero.
fd **(Return Param)**	Returned file descriptor. If *zluopen* is returned with error, this return parameter is undefined.

RETURN VALUE

Routine *zluopen* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include          <zcom/zcomcall.h>

int32                  ierr;
zaddr_type             zaddr;
int                    mode;
int                    fd;

if (ierr = zluopen (&zaddr, mode, &fd)) {
       /* error return code */
}
else {
```

```
        /* good return code */
}
```

NOTES

The ZLU associated device files are kept in the */dev/zcom* directory, and should only be opened by *zluopen*. If they are opened and accessed directly, incorrect results may be returned.

Currently, only device files for program ZLUs are required for the system. They are named as *pzluNNNNN*, where *NNNNN* is a number ranging from 00001 to 99999. The first program ZLU is associated with *pzlu00001*, the second one is associated with *pzlu00002*, etc. (E.g. in a system with 2000 terminal ZLUs, the first program ZLU, i.e. 2001, is associated with the device file */dev/zcom/pzlu00001*.)

Note that these device files are created automatically by the ZCOM subsystem on an as needed basis.

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).
/dev/zcom/pzlu*	Device files associated with program ZLUs.

SEE ALSO

select(2), close(2).

ZMAPR (3X)

NAME

zmapr – Set up ZLU mapping configuration

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zmapr (zap, mzap)

zaddr_type   *zap;
zaddr_type   *mzap;
```

DESCRIPTION

Routine *zmapr* is used to set up an alternate mapping for a ZLU. After the *zmapr* routine has been successfully called, all messages addressed to the source ZLU (**zap*) will be redirected to the destination ZLU (**mzap*).

Both ZLUs must be an existing ZLU such as a:

- program ZLU returned from *zopen*,
- terminal ZLU pre-defined in the TTGEN configuration file,
- mapped ZLU used in previous *zmapr* calls.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address to be mapped.
mzap	ZCOM address to be mapped onto.

NOTES

On a successful return from *zmapr*, accessing the source ZLU is actually directed to the destination ZLU. If the source ZLU was a program ZLU, all messages that were queued on it will be flushed (i.e., returned to the free queue). If it was a terminal ZLU, the linkage to the terminal is lost. Usually, this is not desirable because the previously associated terminal is unusable.

RETURN VALUE

Routine *zmapr* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32          ierr;
zaddr_type     zaddr;
zaddr_type      maddr;

if (ierr = zmapr (&zaddr, &maddr)) {
     /* error return code */
}
else {
     /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZNAME (3X)

NAME

zname – Find ZLU number from ZLU name

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zname (zap, name)

zaddr_type       *zap;
char             *name;
```

DESCRIPTION

When a *zopen* call is issued to allocate a program ZLU, the application assigns a symbolic name to the ZLU. Routine *zname* is used to find the program ZLU number associated with a symbolic name. This allows an application to easily find the program ZLUs of other ZCOM applications running on the ZCOM subsystem. If the symbolic name is not associated with a program ZLU (i.e., is not found in the node specified), the returned ZLU (*zaddr_type.zlu*) is set to zero.

You must set the zap.node field prior to passing the zap parameter to this routine. This will indicate on which node the application we're looking for is located. A value of 0 indicates the local node.

You must call *zinit* in your program before you can invoke *zname* successfully.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.

async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap **(Return param)**	ZCOM address.
name	ZLU symbolic name to search for. The name may be a maximum of **7** characters long, and must be left-justified and null-terminated. Comparison is case sensitive.

RETURN VALUE

Routine *zname* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32        ierr;
zaddr_type   zaddr;
char         name[8];
zsinfo_type sip;

/* zinit() must be called in your program before zname() can be called. */
if (ierr = zinit(&sip)) {
       /* error return code */
    return(ierr);
}

zaddr.node = 0;  /* The local node. */
if (ierr = zname (&zaddr, name)) {
       /* error return code */
}
else {
       /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zopen(3X), makezluname(3X)

ZOPEN (3X)

NAME

zopen – Create ZLU program input queue

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zopen (zap, pflag, name, limit)

zaddr_type       *zap;
uint32           pflag;
char             *name;
uint32           limit;
```

DESCRIPTION

Routine *zopen* allocates a free ZLU as a program input queue. This must be done before a program may receive any messages from terminals or other programs. The calling program's PID is put into the *zowner* field of the *zlu_type* structure.

If a calling program (or thread) terminates without doing a *zclos*(3X) of a ZLU, that ZLU and its resources will remain allocated in the ZCOM system. When the program (or thread) is restarted later on, that ZLU can be re-opened (using the same *name* parameter) without losing its queue contents. The program (or thread) can re-use the ZLU and resume its processing from where it left off. To flush the contents of a re-opened ZLU, use the *zqmve*(3X) routine after the ZLU is opened.

If it is not desirable to re-use an existing ZLU, the ZCOM_ZOPEN_NEW bit of *pflag* can be set. In this case, if the ZLU name already exists, the ZLU is not re-opened and error ZENAME (-11) is returned.

A ZLU can be defined either as a primary or an auxiliary ZLU. Auxiliary ZLUs are used, for example, when a program is receiving messages from multiple sources, so that the messages can be read in a different order from which they were received. All programs must have a primary ZLU defined, because all status messages are returned to the primary ZLU.

In addition, a program's primary ZLU is used as the source ZLU for messages sent by that program (unless overridden by a different ZLU specified in the message header). Refer to the *zsend*(3X) routine for details.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” options to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the NOTES section below for more information on using this routine in a multi-threaded application.

PARAMETERS

zap **(Return Param)**	ZCOM address (node, ZLU number, and checksum). Node contains the local node as defined in the TTGEN configuration file.
pflag	Set to **ZcPRIMARY_ZLU(1)** if this ZLU is to be primary ZLU for this program. Any other value (ZcAUXILIARY_ZLU) defines it as an auxiliary ZLU. Bit ZCOM_ZOPEN_NEW (0x8000) tells the routine to return an error if we are using an existing ZLU.
name	Name to be given to the opened ZLU. The name must be unique within the local node ZLU table. The name may be a maximum of **7** characters long, and must be null-terminated.
limit	Queue size limit. If **0** is specified, then the default queue size limit (defined in the TTGEN configuration file) is used.

RETURN VALUE

Routine *zopen* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

NOTES

The first *zopen* call that a program makes will define the primary ZLU irrespective of the value of *pflag*. If a subsequent *zopen* call is made with *pflag*=1, the primary ZLU definition will be overridden. If multiple *zopen* calls are made with *pflag*=1, then each call will override the previous definition, leaving the primary ZLU defined as the one allocated in the last *zopen* call. It is a good practice to define the primary ZLU in the first call to *zopen* and not to redefine it unless the application program has a good reason for doing so.

In a multi-threaded application, the primary ZLU is shared globally. If a thread opens a primary ZLU, it is effective for all threads.

Routine *zopen* can only allocate a program input ZLU within the local node. The *zaddr* parameter in the call is a return parameter only and contains the number of the local node as well as the ZLU and checksum.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32            ierr;
zaddr_type       zaddr;
uint32           pflag;
char             name[8];
uint32           limit;

if (ierr = zopen (&zaddr, pflag, name, limit)) {
      /* error return code */
}
else {
      /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zsend(3X), zname(3X), zluopen(3X), makezluname(3X)

ZPEEK (3X)

NAME

zpeek – Read data from ZLU without disturbing input queue

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zpeek (zap, mode, mhp, ibuf, len, rlen, rstat)

zaddr_type            *zap;
uint32                mode;
zmhd_type             *mhp;
char                  *ibuf;
int32                 len;
int32                 *rlen;
int32                 *rstat;
```

DESCRIPTION

Routine *zpeek* fetches the next message from the head of a program ZLU input queue. The program has the option to wait (i.e. suspend) if there is no data on the queue specified. The actual length of the data message is returned in *rlen*.

zpeek is similar to *zread*(3X) except that it does not remove the message after retrieving it. Multiple calls to zpeek on the same ZLU will return the same message until *zread*(3X) is called, i.e. it can only look at the first message on the input queue. In addition, the automatic response mechanism is not triggered when fetching a message by *zpeek*, i.e. no response will be sent to the message sender until the message is removed from the input queue with *zread*.

The *zpeek* routine has identical parameters to the *zread* routine. See the *zread* routine for a description of the parameters.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

RETURN VALUE

Routine *zopen* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C (compilation).

SEE ALSO

zread(3X), zluopen(3X)

ZPORT (3X)

NAME

zport – ACC interface card port configuration

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomstatus.h>
#include  <zcom/zcomcall.h> /* if compliled with ANSI C (recommended) */

int32 zport (iftno, portno, rcode, action, cnfg, stat)

uint32       iftno;
uint32       portno;
uint32       rcode;
uint32       action;
zpconf_type  cnfg
int32        *stat;
```

DESCRIPTION

Routine *zport* (*action* 0-3) configures the datacomm ports on an ACC interface card. This configuration is generally specific to the particular hardware implementation of the card and will vary depending on the card type. It is not usually called from application programs. The calling program (or thread) is suspended while this function is executed. All terminals on the port being changed must be **disabled** before the port configuration takes place. For example, use *zcntl* to disable a terminal using *rcode* 8. Examine your TTGEN configuration file to determine which ports are defined for which ACC interface cards and the terminals associated with each port.

Action 4 is used to activate the Port Break Routine of a the protocol specific firmware. Since Port configuration by-passes the normal terminal transmit queues, this function is used as a software interrupt to the protocol firmware to initiate a recovery process whenever there is a protocol problem and the transmit queues are “blocked up”. If the protocol for the specified port does not support this facility, error status 8 is returned in *stat*.

The libraries **libzcom_c.a** and **libpthread.a** must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from multi-threaded application using the POSIX (1003.1c) kernel threads API package. It has the following characteristics when called by multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

iftno ACC MUX interface number.

portno Port number in the ACC interface card:

0.. 7	**For Z7200A, Z7340A and Z7400A ACC interface cards**
0.. 3	For the Z7300A ACC interface card
0.. 1	For the Z7350A ACC interface card

rcode Request code:
1 - Use configuration in interface table
2 - Use data in *cnfg*

action Configuration type:

ZCOM_ZMUXPORT_ALL	(0) - Set all modes (all fields)
ZCOM_ZMUXPORT_CONF	(1) - Set Configuration mode only (Ecode, Mode, Parity)
ZCOM_ZMUXPORT_BAUD	(2) - Set Baud Rate mode only (X.Clk, S.Clk, Baud Rate)
ZCOM_ZMUXPORT_PORT	(3) - Set Port mode only (Pmode, Pmode2)
ZCOM_ZMUXPORT_BRK	(4) - Port Break request

stat
(**Return Param**) Status returned from the call:

PT_OK	(0) - No error
PT_INV_PORT	(1) - Port number out of range
PT_BAD_PARM	(2) - Bad parameter
PT_NOT_DSBL	(3) - Some terminals in this port not disabled
PT_BAD_BAUD	(4) - Bad baud rate
PT_ILL_SCLK	(5) - Illegal clock source

PT_ILL_PMODE	(6) - Illegal port mode
PT_NO_BREAK	(7) - Port failed self test
PT_NO_BREAK	(8) - BREAK routine not installed
PT_BAD_XCLK	(9) - Clock multiplier not compatible with mode
PT_NO_PLL	(10) - No DPLL available at this baud rate
PT_TOO_FAST	(11) - Async mode not available for baud > 38400

The status message text may be fetched using *zcomstatus*(3X) (*req* = ZCOM_MRQCODE_PORT, or 11).

cnfg Configuration data (32 bits)

The *cnfg* parameter contains the following information for the Z7200A, Z7400A, Z7340A and Z7350A ACC interface cards:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Ecode		Sync		Mode		Parity		X. Clk		S. Clk		Baud 1			
Pmode		Pmode2		Baud 2				0 = Not Used							

Ecode	*Value*	*Encoding modes*
	0 0	NRZ
	0 1	NRZI
	1 0	FM1 (transition = 1)
	1 1	FM0 (transition = 0)

Ecode	*Value*	*Sync mode select*
	0 0	8-Bit sync (monosync)
	0 1	16-Bit sync (bisync)
	1 0	SDLC Mode
	1 1	External sync

Ecode	*Value*	*Operating mode*
	0 0	Sync mode
	0 1	Async 1 stop
	1 0	Async 1.5 stop bits
	1 1	Async 2 stop bits

Parity	*Value*	*Parity select*
	0 0	No Parity
	0 1	Odd parity
	1 0	No Parity
	1 1	Even parity

X.Clk	*Value*	*Clock multiplier*
	0 0	X 1
	0 1	X 16
	1 0	X 32
	1 1	X 64

S.Clk	*Value*	*Clock source*
	0 0	External clock
	0 1	Internal clock from BRG
	1 0	X.21 clock source
	1 1	DPPL output (must use BRG as source)

The baud rate is split between two 4-bit parameters (for compatibility reasons). If the Baud 1 parameter is 0, then Baud 2 is used. Rates listed in **bold** are not available on the Z7350A or Z7340A ACC interface cards. The Z7200A card does not support baud rates above 76800 and the Z7400A card does not support rates above 128000.

Baud 1	*Value*	*Rate*	*Value*	*Rate*
	0000	Use Baud2	1000	19,200
	0001	**300**	1001	38,400
	0010	600	1010	48,000
	0011	1,200	1011	57,600
	0100	2,400	1100	76,800
	0101	4,800	1101	64,000
	0110	9,600	1110	128,000
	0111	14,400	1111	256,000

Baud 2	*Value*	*Rate*	*Value*	*Rate*
	0000	**150**	1000	768,000
	0001	56,000	1001	1,024,000
	0010	153,600	1010	1,228,800
	0011	307,200	1011	1,536,000
	0100	192,000	1100	1,544,000
	1010	614,400	1101	2,048,000
	0110	384,000	1110	Reserved
	0111	512,000	1111	Reserved

Pmode	*PmodeValue*	*PmodeValue*	*Port mode*

0 0	0 0	RS232 Mode
0 1	0 0	RS422 Mode
1 0	0 0	Loopback Mode (Tristate)
1 1i	0 0	X.21/V.11 Mode
0 0	0 1	V.35 (Z7340A and Z7350A only)
0 0	1 0	RS-449 (Z7340A and Z7350A only)
0 0	1 1	V.36 (Z7340A and Z7350A only)
All Other Values		Reserved

The Z7300A or Z7330A E1/T1 card uses the following configuration:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Ecode		Fmode				0 = not used				S. Clk		0 = not used			
Pmode				0 = not used				0 = Not Used							

Ecode	*Value*	*Encoding modes*
	0 0	HDB3 - High Density Bipolar/3 zeros
	0 1	B8ZS - Bipolar/8 Zero Substitution
	1 0	AMI - Alternating Mark Inversion (AMI is not supported)
	1 1	Not Defined

Fmode	*Value*	Frame Mode parameter
	0000	T1: F4 - 4-Frame Multiframe
	0001	T1: SF - 12-Frame Multiframe
	0010	T1: ESF - Extended Superframe

	0011	T1: F72 - Remote Switch Mode
	0100	E1: DF - Doubleframe
	0101	E1: MF - CRC Multiframe
	0110	Transparent (voice) mode
S.Clk	*Value*	Clock source
	0 0	External clock (slave)
	0 1	Internal clock from BRG (master)
	1 0	External RxC only (Exin)
	1 1	DPPL output (use BRG as source)

Pmode	*Value*	*Port Mode*
	0000	Reserved
	....	
	0100	Reserved
	0101	E1 Twisted pair DB9 120 ohm
	0110	E1 Coax BNC 75 ohm
	0111	T1 Twisted pair RJ45 100 ohm
	1000	Loopback Mode (Tristate)
	1001	E1 Twisted pair RJ45 100 ohm
	1010	T1 Twisted pair DB9 120 ohm
	1011	T1 Coax BNC 75 ohm
	1100	Reserved
	...	
	1111	Reserved

NOTES

1. Notice that it is possible to change only one byte of the configuration word by using an appropriate 'action' parameter value. In this case, the other bytes remain unaffected. The new parameters remain in force until changed by another call or until the ZCOM subsystem is reinitialized. Normally, an application program should not be required to change the port configuration.
2. To configure the timeslot and subchannel specification information on the ACC E1/T1 card, use the *zconfig()* function.
3. The following is the *zpconf_type* data structure definition (from *<zcom/zcomsys.h>*). This shows how the *cnfg* data can be referenced by individual bit fields (*zpconf_type.bits.<field>*), by individual bytes (*zpconf_type.pconfig[<bytenum>]*), or by its entire 32-bit value (*zpconf_type.pcval*).

```
/*-- Port Configuration format --*/
 typedef union {
     struct {        /* Port config for all non-E1/T1 cards. */
         unsigned int ecode   : 2;   /* Encoding modes */
         unsigned int sync    : 2;   /* Sync mode select */
         unsigned int mode    : 2;   /* Operating mode */
         unsigned int parity  : 2;   /* Parity select */
         unsigned int xclock  : 2;   /* Clock multiplier */
         unsigned int sclock  : 2;   /* Clock source */
         unsigned int baud    : 4;   /* Baud rate value */
         unsigned int pmode   : 2;   /* Port mode select - original */
         unsigned int pmode2  : 2;   /* Port mode select - extended */
         unsigned int baud2   : 4;   /* Extended baud rate value */
         unsigned int spare   : 8;   /* Reserved, must be 0 */
     } bits;
    struct {         /* E1/T1 port configuration (Z7300A) */
          unsigned int ecode  : 2;    /* Encoding modes */
          unsigned int frmode : 4;    /* Framing mode */
          unsigned int set0_1 : 4;    /* Unused */
          unsigned int sclk   : 2;    /* Clock source */
          unsigned int set0_2 : 4;    /* Unused */
          unsigned int pmode  : 4;    /* Port Mode */
          unsigned int set0_3 : 12;   /* Unused */
     } e1t1_bits;
     int32 pcval;        /* Must be signed, -1 means bad config */
     uint8 pconfig[4];   /* 4 configuration bytes */
 } zpconf_type;
```

RETURN VALUE

Routine *zport* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

When *zport* returns a zero error code, the return status *stat* indicates the completion status: zero means successful, while non-zero values indicate the reason for the problem. Note that status code 4, 5, 6, and 7 implies the associated port will now be inoperative. Another *zport* configuration request must be done to recover it.

EXAMPLE This example puts an E1/T1 port into “loopback” (tristate) mode.

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>
 int32          ierr;
 uint32         iftno;
 uint32         portno;
 uint32         rcode;
 uint32         action;
 zpconf_type    cnfg;
 int32          stat;

iftno  = 1;     /* ACC MUX number 1 in TTGEN configuration file. */
 portno = 2;     /* Port #2 */
 rcode  = 2;     /* We are providing the configuration in "cnfg". */
 action = ZCOM_ZMUXPORT_PORT;  /* Set only the port mode (Pmode, Pmode2) */
 cnfg.e1t1_bits.pmode = 0x8;
 if (ierr = \fBzport\fR (iftno, portno, rcode, action, cnfg, &stat)) {
      /* error return code */ } else if (stat)
 {
      /* error return code */ } else {
          /* good return code */
 }
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO zconfig(3X)

ZPTUP (3X)

NAME

zptup - Update physical terminal table user area

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zptup (zap, ibuf, len, offset)

zaddr_type   *zap;
char         *ibuf;
int32        len;
int32        offset;
```

DESCRIPTION

Routine *zptup* is used to update information in the user area of the physical terminal table. This user area contains transmit and receive message counters as well as error counters and special configuration information specific to the protocol being used. For example, if the terminal is using X.25 packet switching, it would hold X.25 related details. See under the section on the physical terminal tables.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address of the physical terminal table to update.
ibuf	Data buffer holding the information to place into the PTT.
len	Number of bytes of data to copy from *ibuf* into the PTT.
offset	Byte offset within the physical terminal table from where the update should begin.

RETURN VALUE

Routine *zptup* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32          ierr;
zaddr_type     zaddr;
char           ibuf[size]; /* where size is user-determined; must be >= len */
int32          len;
int32          offset;

if (ierr = zptup (&zaddr, ibuf, len, offset)) {
          /* error return code */
}
else {
          /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZQMVE (3X)

NAME

zqmve – Move message from one ZLU to another

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zqmve (zap, dzap)

zaddr_type  *zap;
zaddr_type  *dzap;
```

DESCRIPTION

Routine *zqmve* is used to transfer the first message from one ZLU queue to the end of a second ZLU queue by changing the queue pointers (without actually moving the data). If the destination queue is not specified (i.e., *dzap* = NULL), the message will be flushed, which means the message buffer is returned to the free data buffer pool.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	ZCOM address from which to get the message.
dzap	Destination ZCOM address (if NULL, then the message will be flushed).

RETURN VALUE

Routine *zqmve* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
Example 1:
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32          ierr;
zaddr_type     zaddr;
zaddr_type     daddr;

if (ierr = zqmve (&zaddr, &daddr)) {
          /* error return code */
}
else {
          /* good return code */
}


Example 2: - (how to flush)
while(zqmve(&zaddr, NULL) == NULL);
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZQSZE (3X)

NAME

zqsze – Read ZLU input queue size

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zqsze (zap, zqhdp)

zaddr_type  *zap;
zqhd_type   *zqhdp;
```

DESCRIPTION

Routine *zqsze* will return the queue header information, which includes the number of messages, i.e., the number of buffers (*zqhd_type->qnmsg*) queued to the specified program ZLU. An error is returned if the ZLU is not a program ZLU.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

za ZCOM address
p

zqhdp **(Return param)**

ZCOM queue header data structure. This structure has the following definition:

```
typedef struct {
    uint32    qnmsg;       /* Number of msgs on queue */
    uint32    qlimit;      /* Max # of msgs allowed on queue */
    zbhd_type *qhead;      /* Pointer to 1st msg on queue */
    zbhd_type *qtail;      /* Pointer to last msg on queue */
    uint32    qmmax;       /* Historical max # of msgs on queue */
    uint32    qtmsg;       /* Total # of msgs through queue */
    uint32    qbytes;      /* Number of bytes on queue */
    uint16    qwaiter;     /* Number of waiters on this queue */
    uint16    qflag;       /* Queue flags */
    uint32    qfdata;      /* Queue function data */
    uint32    spare1;      /* Reserved, not used */
    void      (*qfunc)();  /* Queue function to call */
}   zqhd_type;
```

Note that in 32-bit mode, 32-bit padding fields are inserted before the pointer fields. These are to maintain the same layout in both 32/64-bit modes. Such padding fields are not shown here and also should not be referenced directly.

RETURN VALUE Routine *zqsze* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32          ierr;
zaddr_type     zaddr;
zqhd_type      zqhd;

if (ierr = zqsze (&zaddr, &zqhd)) {
       /* error return code */
}
else {
       /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZREAD (3X)

NAME

zread – Read from ZLU

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zread (zap, mode, mhp, ibuf, len, rlen, rstat)

zaddr_type       *zap;
uint32           mode;
zmhd_type        *mhp;
char             *ibuf;
int32            len;
int32            *rlen;
int32            *rstat;
```

DESCRIPTION

Routine *zread* fetches the next message from the head of a program ZLU queue. The program has the option to wait (i.e., suspend) if there is no data on the queue specified. The actual length of the data message is returned in *rlen*.

This routine is similar to *zpeek*(3X) except it removes the message after fetching it.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.

fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap	Pointer to program ZCOM address. Source to get input messages.
mode	ZcREAD_W_WAIT (0) - Read with wait. The ztimr() call can be used to set a timeout period that will cause the zread() call to unsuspend and continue after the timeout period has elapsed. ZcREAD_WO_WAIT (1) - Read without wait. **Note**: If *mode* = **0**, the program is suspended until there is a message on the input queue. If *mode* = **1**, and there is no message on the queue, error ZENODATA (-24) is returned. Otherwise, the message is placed in *ibuf*. If bit ZCOM_ZREAD_SHARE (0x2000) is set, the input may be shared by multiple processes. This allows multiple processes to read (and wait) on the same ZLU. When a message arrives, only *one* of the waiting processes will return with the data. If bit ZCOM_ZREAD_QUIET (0x4000) is set, the automatic response mechanism is suppressed. Normally, when a message is sent (e.g., by zsend(3X)) with a mode requesting response, a response is generated automatically when the message is read by *zread*. This bit is used to suppress this response, and the caller is responsible for checking the request mode and sending a response back to the originator if necessary. This feature is intended for multiplexing programs and should be used with care. This bit is not used in *zpeek*(3X).
mhp **(Return Param)**	Pointer to message header. It is the message header received with the data buffer. NULL may be specified if the caller does not require the returned message header. It is described in the subsection on 'Message Header' in the *Multiprotocol ACC Programmers' Reference Guide*. See the data structure *zmhd_type* in *zcomsys.h* for more detail.
ibuf **(Return Param)**	Buffer for the received data.
len	Maximum length in bytes of the buffer *ibuf*.

rlen **(Return Param)** Actual data length. NULL may be specified if the caller does not require the returned message length. Note that *rlen* may exceed *len*. This indicates the received data is truncated to just fill up *ibuf*.

rstat **(Return Param)** Return status.

0 - no error

-ve - ZCOM error code

+ve - terminal error status

It is meaningful only if the returned message is a response or terminal message. For other message types, it is always returned with zero. NULL may be specified if the caller does not require this return status.

The return status *rstat* and the request code *mhp->mrq.mrqcode* may be used to retrieve a status message by calling *zcomstatus*(3X).

NOTES

1. If *zsend*(3X) was used to send data using modes 1, 2, 5, or 7, then the input ZLU queue may contain response status messages. If the sent message could not be delivered by the driver (e.g., data corruption in the host system), an error response is returned to the sender's primary ZLU. When such an error response is picked up by *zread*, the function returns with zero (indicating no problem in reading the input ZLU), the returned message type is set (e.g., ZCOM_MSTYPE_RSLT) and the return status will carry a negative ZCOM error code. If the message could not be delivered by the interface (e.g., a problem with the terminal line), the returned response will have a positive ZCOM status code indicating the problem.
2. If the received message (or response) is from a terminal, the request code is returned in *mhp->mrq.mrqcode*. The request code and the returned status may be used to retrieve an appropriate status/error message by *zcomstatus*(3X). Refer to Appendix B ZCOM Status Codes of the *Multiprotocol ACC Programmers' Reference Guide* for more information. The *mhp->mrq.mrqcode* field contains 0 if the message is from a program or is a System Event Message (SEM).

3. A System Event Message (SEM) is generated by the ZCOM subsystem in response to special events that occur within the ZCOM subsystem. A program may choose to receive event messages by supplying its primary ZLU in a *zevent_rcvr* call. All ZCOM Event messages are divided into a header and event specific data section. The header is defined for all types of Event messages while the data portion depends on the ZCOM Event Message type. Refer to the subsection 'System Event Messages' in the *Multiprotocol ACC Programmers' Reference Guide* for more information. The SEM's structure is defined in */opt/acc/include/zcom/zcomsys.h*.
4. The tag parameters (*mhp->mid.mtagw1* and *mtagw2*) are the tag parameters from the originating zsend(3X) call.

5. The type of message received is returned in *mhp->mid.mstype*. Its values are as follows (defined in */opt/acc/include/zcom/zcomsys.h*):

```
#define  ZCOM_MSTYPE_REMOTE     0x08        /* Mask for remote message bit */
#define  ZCOM_MSTYPE____0        0          /* Undefined type */
#define  ZCOM_MSTYPE_MSLT        1          /* From local terminal */
#define  ZCOM_MSTYPE_LPLP        2          /* Local program to program */
#define  ZCOM_MSTYPE_TOLZ        3          /* Timer message for local ZLU */
#define  ZCOM_MSTYPE_LPLT        4          /* Local program to terminal */
#define  ZCOM_MSTYPE_LSEM        5          /* Local system event message */
#define  ZCOM_MSTYPE_RSLT        6          /* Response from local terminal */
#define  ZCOM_MSTYPE_RSLP        7          /* Response from local program */
#define  ZCOM_MSTYPE____8        8          /* Undefined type */
#define  ZCOM_MSTYPE_MSRT        9          /* Remote terminal message */
#define  ZCOM_MSTYPE_RPLP        10         /* Remote program to program */
#define  ZCOM_MSTYPE_TORZ        11         /* Timer message for remote ZLU */
#define  ZCOM_MSTYPE_RPLT        12         /* Remote program to terminal */
#define  ZCOM_MSTYPE_RSEM        13         /* Remote system event message */
#define  ZCOM_MSTYPE_RSRT        14         /* Response from remote terminal */
#define  ZCOM_MSTYPE_RSRP        15         /* Response from remote program */
```

RETURN VALUE

Routine *zread* returns **0** if the call is successful. In addition, for a response message, a **non-zero** return status *rstat* indicates there was a problem with the previous *zsend*(3X) call. A **non-zero** return status with a terminal message or response indicates there was a problem with the terminal. For other message types, the return status is always **0**.

If the *zread* call itself failed, a **non-zero** error code is returned, and the other return parameters do not contain valid values. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include   <zcom/zcomsys.h>
#include   <zcom/zcomcall.h>

int32            ierr;
zaddr_type       zaddr;
uint32           mode;
zmhd_type        zmhd;
char             ibuf[size]; /* where size is user-determined; must be >= len */
int32            len;
int32            rlen;
int32            rstat;

if (ierr = zread(&zaddr, mode, &zmhd, ibuf, len, &rlen, &rstat)) {
```

```
        /* error return code */
}
else {
        /* good return code */
}
```

NOTE

Check *zmhd.mid.mstype* to determine the message type. For response or terminal messages, check for **non-zero** *rstat* to see if there was any problem with the previous *zsend*(3X) or with the terminal.

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zevent_rcvr(3X), zpeek(3X), zset_rcvr(3X), zsend(3X), ztimr(3X), zcomstatus(3X).

ZRNTIMER (3X)

NAME

zrntimer – Set timeout for remote node access

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zrntimer (node, time)

int32       node;
int32       time;
```

DESCRIPTION

Routine *zrntimer* is used to change the ZCOM request timeout value for requests sent to a remote ZCOM system *node*. The new timeout only affects the calling program, and is effective for all subsequent requests for that node until another *zrntimer* call changes the timeout or whenever the program is restarted. In the latter case, the timeout is reset to the default value as defined in the TTGEN configuration file.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

not a cancellation point	It has no cancellation point and does not call any function with cancellation point. Thread cancellation will not occur when a thread is calling this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

See the NOTES section below for more information on using this routine in a multi-threaded application.

PARAMETERS

node Remote node number where timeout is set. A value of zero means setting the timeout values for ALL remote nodes.

time Timeout value in seconds (1-65535). Zero or negative values are not allowed.

RETURN VAULE

Routine *zrntimer* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

NOTES

Setting the timeout value for the local node will complete without error. However, the timeout value will have no effect because ZCOM does not timeout local requests (i.e. a program may be suspended forever in order to complete its local request).

An internal table is maintained to keep track of the timeout values for each remote node. The table is initialized with the default values defined in the TTGEN configuration file. If an unknown node number is specified, it is rejected with error -6 (ZENODE).

In a multi-threaded application, all the threads share the single timeout value table. Hence, when a thread sets a node timeout value, the new value will be effective in all the other threads.

Although a timeout value of one (1) is a legal value, a minimum value of two (2) is used in order to avoid false alarms because the internal timeout processing is activated on 1-second intervals.

EXAMPLE

```
#include      <zcom/zcomsys.h>
#include      <zcom/zcomcall.h>

int32         ierr;
int32         node;
int32         time;

if (ierr = zrntimer (node, time)) {
       /* error return code */
}
else {
```

```
        /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZSEND (3X)

NAME

zsend – Send data buffer to a ZLU

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zsend (zap, mode, mhp, ibuf, len, rstat)

zaddr_type       *zap;
uint32           mode;
zmhd_type        *mhp;
char             *ibuf;
int32            len
int32            *rstat;
```

DESCRIPTION

Routine *zsend* sends a buffer to the specified ZLU. The zcntl(3X) routine is used for sending control messages to a terminal ZLU.

Zero length buffers are allowed (usually sent to other program ZLUs). For example, a zero length send may be used to signify a special event with some information in the header tag words, *mhp->mid.ztagw1* and *mhp->mid.ztagw2*.

The response returned from zsend is through the primary ZLU unless a different source ZLU is specified in the header (*mhp->mid.mzsrce*).

Note that a program must call zopen() to allocate a primary or auxiliary input ZLU before calling the zsend() routine. This must be done even if the program does not need an input ZLU, for example, if it is not doing any zread() calls and is only doing zsend() calls with mode = ZcMODE_NO_WAIT (0) (send no-wait) where it does not care about the return status.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap Pointer to ZCOM address. The destination to where the data is sent.

mode Completion control:

ZcMODE_NO_WAIT	(0) - Send no wait
ZcMODE_RPT_ERRS	(1) - Send no wait, report errors
ZcMODE_DEF_STATUS	(2) - Send no wait, definite status
ZcMODE_RPT_ERRS_WBUF	(5) - Send no wait, report errors with buffer
ZcMODE_DEF_STATUS_WBUF	(7) - Send no wait, definite status and buffer
ZcMODE_WAIT	(8) - Send and await status in *rstat*

Some bits (when set) in this mode parameter can cause *zsend* to behave differently. Refer to the NOTES section below for details.

mhp Pointer to message header. It contains some data fields to be sent with the data message. See the NOTES section below for what fields must be specified. If NULL is specified, default values are used.

For its structure, refer to the subsection on 'Message Header' in the *Multiprotocol ACC Programmers' Reference Guide*.

ibuf Buffer to send (character array).

len Length in bytes to be sent.

rstat **(Return Param)** Return status. Only used for requests issued with 'wait', i.e., *mode* = **8**. NULL may be used if the return status is not required.

NOTES

If *mhp* is specified (i.e., a non-NULL pointer), some header fields must be set up by the caller because they are extracted and sent with the data buffer. These fields are:

mhp->mid.mzsrce	- source ZCOM address (ZLU)
mhp->mid.mtagw1	- header tag word 1
mhp->mid.mtagw2	- header tag word 2
mhp->mrq.mrqtag	- protocol tag parameter (for terminal ZLU only)

The other header fields are set up by the ZCOM subsystem exclusively.

The source ZCOM address field provides a way to redirect the response to a different ZLU rather than the primary ZLU. The Primary ZLU is used if it is zero. The two header tag words provide a means of identifying the response in the case where a “no wait” mode is used. The protocol tag parameter controls special options supported by the protocol module (protocol specific). Refer to the relevant *Protocol Manual* for its use.

It is important to zero out undesirable fields to avoid unexpected results. For example, if the caller only wants to specify the two header tag words, then the source ZLU and protocol tag must be zeroed out. If NULL is specified for *mhp*, *zsend* defaults the source ZLU to the primary ZLU and the rest to zeros.

If bit ZCOM_ZSEND_MHD (0x8000) of *mode* is set, *zsend* allows more header fields to be specified by the caller, hence having more control in the message sent. This implies *mhp* must be specified, i.e., non-NULL. The header fields that must be set up by the caller are:

mhp->mid.mstype	- message type
mhp->mid.msresp	- message response code
mhp->mid.mzdest	- destination ZCOM address
mhp->mid.mzsrce	- source ZCOM address
mhp->mid.mtagw1	- header tag word 1
mhp->mid.mtagw2	- header tag word 2
mhp->mid.mzaux1	- auxiliary ZCOM address

mhp->mrq.mrqstat	- returned status (for response messages)
mhp->mrq.mrqtag	- protocol tag value (for terminal ZLU only)

If Bit ZCOM_ZSEND_NOMX (0x4000) of *mode* is set and the request is to a outbound-multiplexed terminal ZLU, the outbound multiplexing is over-ridden. This results in the data being sent directly to the terminal (instead of the multiplexing program).

The ZCOM_ZSEND_MHD and ZCOM_ZSEND_NOMX bits are intended for use by multiplexing programs to intercept terminal messages and build specific messages or responses as required.

They can cause strange ZCOM subsystem behavior if used improperly; hence, they must be used with caution.

If bit ZCOM_ZSEND_LPR (0x0080) or ZCOM_ZSEND_XPS (0x0040) is set and the request is to a terminal, then the message will be queued on the **low** priority or **express** transmit queue, respectively. Otherwise, it defaults to the **high** priority queue.

The different transmit queues allow the caller to have some control over the order of messages being sent. The express queue is always processed first and is not bounded by queue limit. It is intended for sending critical control data and should never be used for normal data. The high priority queue is subject to queue limit checks and is intended for sending normal data. The low priority queue is processed only after all express and high priority messages are sent and is also bounded by queue limit checks. It is intended for sending data in a background mode. For a busy terminal, low priority messages will not be sent until all its higher priority activities are completed.

Apart from the special bits, the *mode* value indicates the requested completion status of the write:

* *Mode*=**0** (ZcMODE_NO_WAIT - send no wait)

The message to be sent is moved to a buffer and the caller continues executing without waiting for return status. If this mode is used, the program is responsible for controlling the flow of messages. If the calling program (or thread) uses too many buffers it will be suspended until some of the buffers have been sent.

Even if the caller uses this mode and does not care about the return status, it must still call *zopen()* to allocate a primary or auxiliary input ZLU before making the *zsend* call.

* *Mode*=**1** (ZcMODE_RPT_ERRS - send no wait, report errors)

The message is transmitted without waiting for completion. A status response will be returned on the primary ZLU of the calling program, only if transmission errors occur. The response is retrieved by using a *zread*(3X) call; it will have a message type of six (6) or fourteen (14). Only the status is returned (in the *rstat* parameter of *zread*(3X)). The data buffer is discarded. This mode is used if the caller is only interested in knowing about unsuccessful zsend (transmit) requests.

* *Mode*=**2** (ZcMODE_DEF_STATUS - send no wait, definite status)

There is a definite response for every message sent with this mode. The response is returned to the application's program ZLU after the message has been successfully sent or upon a transmit error. The response is retrieved by using the *zread*(3X) call; it will have a message type of six (6) or fourteen (14). Only the status is returned (in the *rstat* parameter of *zread*(3X)). The data buffer is discarded. This mode is useful in sending a large number of messages through ZCOM with flow control.

* *Mode*=**5** (ZcMODE_RPT_ERRS_WBUF - send no wait, report errors with buffer)

Similar to Mode 1, however the data buffer is returned along with the status if any transmission errors occur. In this way the data buffer can easily be rerouted, resent or logged.

* *Mode*=**7** (ZcMODE_DEF_STATUS_WBUF - send no wait, definite status with buffer)

Similar to Mode 2, however both the status and the data buffer are returned with the response. This enables a message to be sent to multiple ZLUs in a particular order, without the program having to store the message temporarily.

* *Mode*=**8** (ZcMODE_WAIT - send and await status)

The return status *rstat* indicates whether there is any problem with the send (zero means successful). The calling program (or thread) is suspended by this call while the transmission is in progress. In a multi-threaded application, only the calling thread will be suspended, the other threads are not affected. When transmission is completed, the call is returned with the return status in *rstat*. This mode is used if the caller cannot continue unless the data message is completely sent.

This mode of transmission is not only slower but also has a higher processing overhead than the no wait options described above. This mode does not provide for any timeouts at this level, i.e., it returns only if the protocol completes or times out the transmission. It should therefore be used with caution.

The following data structures are used for the *zsend* call. See */opt/acc/include/zcom/zcomsys.h* for further details.

```
/*-- ZCOM Message ID Header --*/
  typedef struct {
    uint8          mstype;     /* Message Type */
    uint8          msresp;     /* Message response code */
    zaddr_type     mzdest;     /* Destination zcom addr  */
    zaddr_type     mzsrce;     /* Source zcom addr */
    uint16         mtagw1;     /* Message tag word 1 */
    uint16         mtagw2;     /* Message tag word 2 */
    zaddr_type     mzaux1;     /* Auxiliary zcom addr */
  } zmid_type;

/* Mask and values for message type */
    #define  ZCOM_MSTYPE_REMOTE    0x08   /* Mask for remote message bit */
    #define  ZCOM_MSTYPE____0      0      /* Undefined type */
    #define  ZCOM_MSTYPE_MSLT      1      /* From local terminal */
    #define  ZCOM_MSTYPE_LPLP      2      /* Local program to program */
    #define  ZCOM_MSTYPE_TOLZ      3      /* Timer message for local ZLU */
    #define  ZCOM_MSTYPE_LPLT      4      /* Local program to terminal */
    #define  ZCOM_MSTYPE_LSEM      5      /* Local system event message */
```

```
    #define  ZCOM_MSTYPE_RSLT      6      /* Response from local terminal */
    #define  ZCOM_MSTYPE_RSLP      7      /* Response from local program */
    #define  ZCOM_MSTYPE____8      8      /* Undefined type */
    #define  ZCOM_MSTYPE_MSRT      9      /* Remote terminal message */
    #define  ZCOM_MSTYPE_RPLP      10     /* Remote program to program */
    #define  ZCOM_MSTYPE_TORZ      11     /* Timer message for remote ZLU */
    #define  ZCOM_MSTYPE_RPLT      12     /* Remote program to terminal */
    #define  ZCOM_MSTYPE_RSEM      13     /* Remote system event message */
    #define  ZCOM_MSTYPE_RSRT      14     /* Response from remote terminal */
    #define  ZCOM_MSTYPE_RSRP      15     /* Response from remote program */

/* Mask of message response code */
    #define  ZCOM_MSRESP_LPR       0x80   /* Low priority transmit message */
    #define  ZCOM_MSRESP_XPS       0x40   /* Express transmit message */
    #define  ZCOM_MSRESP_PGW       0x08   /* Program is waiting for message */
    #define  ZCOM_MSRESP_BFR       0x04   /* Response with buffer and status */
    #define  ZCOM_MSRESP_DEF       0x02   /* Definite response */
    #define  ZCOM_MSRESP_ERR       0x01   /* Response if error */

/*-- ZCOM Message Request Header --*/
typedef struct {
    uint8        mrqcode;         /* Request code */
    uint8        mrqterm;         /* Terminal no. */
    uint8        mrqstat;         /* Request status */
    uint8        mrqtag;          /* Tag parameter */
    uint16       mrqlen;          /* Data length */
}  zmrq_type;

/* Message request code values */
    #define ZCOM_MRQCODE_UND       0     /* Undefined type */
    #define ZCOM_MRQCODE_READ      1     /* Terminal input */
    #define ZCOM_MRQCODE_WRITE     2     /* Terminal write */
    #define ZCOM_MRQCODE_CNTWR     3     /* Terminal write */
    #define ZCOM_MRQCODE_STDT      4     /* INTERNAL USE: Status transfer */
    #define ZCOM_MRQCODE_STATUS    5     /* Unsolicited status change */
    #define ZCOM_MRQCODE_TERM      6     /* Set terminal parameters */
    #define ZCOM_MRQCODE_ENB       7     /* Terminal enable */
    #define ZCOM_MRQCODE_DSB       8     /* Terminal disable */
    #define ZCOM_MRQCODE_ACT       9     /* Terminal activate */
    #define ZCOM_MRQCODE_DEA       10    /* Terminal deactivate */
    #define ZCOM_MRQCODE_DEL       11    /* Terminal delete */
    #define ZCOM_MRQCODE_PORT      14    /* Port configuration */
    #define ZCOM_MRQCODE_MULTI      15    /* Multi-completion RXDT req code. */

     /*-- ZCOM Message Header --*/
typedef struct {
        zmid_type    mid;             /* Message ID header */
        zmrq_type    mrq;             /* Message request block */
}  zmhd_type;
```

RETURN VALUE

Routine *zsend* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

If *mode* ZcMODE_WAIT is used and *zsend* returns with zero, *rstat* contains the return status: **0** means successful, **non-zero** means there was a problem with the send. *zcomstatus*(3X) may be used to retrieve a status message, using request code ZCOM_MRQCODE_WRITE and *rstat*.

EXAMPLE

```
#include   <zcom/zcomsys.h>
#include   <zcom/zcomcall.h>

int32           ierr;
zaddr_type      zaddr;
uint32          mode;
zmhd_type       zmhd;
char            ibuf[size]; /* where size is user-determined; must be >= len */
int32           len;
int32           waitstat;

if (ierr = zsend (&zaddr, mode, &zmhd, ibuf, len, &waitstat)) {
   /* error return code */
}
else {
   /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zcomstatus(3X), zcntl(3X), zread(3X), zopen(3X).

ZSETQL (3X)

NAME

zsetql – Set buffer queue limit

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zsetql (zap, mode, qnum, limit)

zaddr_type        *zap;
uint32            mode;
uint32            qnum;
uint32            limit;
```

DESCRIPTION

Routine *zsetql* changes the pre-set queue limit of a buffer queue (the pre-set value is configured in the TTGEN configuration file). This allows the caller to adjust the input/output characteristics of some of the ZLUs in the ZCOM subsystem without restarting the ZCOM subsystem or affecting other ZLUs.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap ZCOM address.

mode Type of queue:

ZcPROGRAM_Q	(0) - Program input queue
ZcLOGICAL_TERM_Q	(1) - Logical terminal queue
ZcPHYSICAL_TERM_Q	(2) - Physical terminal queue

If bit ZCOM_QLIM_MFLAG (0x8000) is set, the queue is changed to use a message limit. If bit ZCOM_QLIM_BFLAG (0x4000) is set, the queue is changed to use a byte limit. If neither are set, the limit mode is unchanged. They should not be set together.

qnum Queue number. Depending on *mode*, it specifies which queue to set:

mode 0 -	(ZcPROGRAM_Q) *qnum* parameter not used
mode 1 -	(ZcLOGICAL_TERM_Q) Data queue number
	0 = Sub-packet holding queue
	1 = Data holding queue A
	2 = Data holding queue B, etc.
mode 2 -	(ZcPHYSICAL_TERM_Q) Data queue number
	0 = High priority transmit queue
	1 = Low priority transmit queue
	2 = Unacknowledged transmit queue
	3 = Express transmit queue

limit New queue limit. Allowable range is 0 to ZCOM_MAXQLIM (0x7FFFFFFF).

RETURN VALUE

Routine *zsetql* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

NOTES

1. Queue limit does not affect data messages that are already in a queue. If the new limit is below the amount of queued data, no new data is accepted until the queued data drops below the limit. Setting a queue limit to zero will prevent the queue from accepting new messages. The queue is still readable, but new data can't be added to the queue.
2. Under the current implementation, the terminal unacknowledged queue uses a "byte limit", i.e. the limit for this queue indicates the maximum number of bytes allowed on the queue. The other queues use a "message limit", which means the maximum number of messages allowed on the queue. However, the limit type may be changed by setting the ZCOM_QLIM_MFLAG or ZCOM_QLIM_BFLAG bit in the *mode* parameter.
3. The system initializes the express transmit queue (*mode* 2, *qnum* 3) with the maximum limit (0x7FFFFFFF). A low limit may affect critical data getting to the interface card and result in terminal control problems. Therefore, it is strongly recommended that the limit not be changed unless it is absolutely necessary. With some protocol options (e.g. X.25), changing this limit is unsupported!

EXAMPLE

```
#include <zcom/zcomsys.h>
#include <zcom/zcomcall.h>

int32       ierr;
zaddr_type  zaddr;
uint32      mode;
uint32      qnum;
uint32      limit;

if (ierr = zsetql (&zaddr, mode, qnum, limit)) {
   /* error return code */
}
else {
   /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM subsystem general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

ZSET_RCVR (3X)

NAME

zset_rcvr – Set up program ZLU as message receiver for a terminal ZLU

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 zset_rcvr (action, zap, mode, rzap)

int             action;
zaddr_type      *zap;
uint32          mode;
zaddr_type      *rzap;
```

DESCRIPTION

The *zset_rcvr()* routine is used to establish where incoming data from a terminal ZLU is to be queued. When *zset_rcvr()* is called, the **rzap** parameter is used to supply the program ZLU to queue the data on. (Note: An application's program ZLU is returned by the *zopen()* routine.)

The type of data queued is selected using the *mode* parameter. You may choose to receive data/status messages, and/or control messages. The location (Virtual Circuit/device/terminal/etc) from which you want to receive the data is specified by the *zap* parameter. An *action* parameter value of ZcADD_PRIMARY sets the primary receiver.

Once you have called *zset_rcvr()*, the application program can issue *zread()* requests to retrieve any data that may have arrived from the selected terminal ZLU.

An application program that issues a *zset_rcvr()* call, as described above, is the primary receiver of the specified messages. Another application program can take over being the primary receiver by issuing a *zset_rcvr()* request, using the same mode and rzap parameters, with the action code also set to ZcADD_PRIMARY. In this situation, data is no longer queued to the original application.

The *zset_rcvr()* routine can also be used to allow messages to be received by more than one application program at a time. This feature is selected by using an *action* parameter value of ZcADD_SHARED. When ZcADD_SHARED is used, it specifies that the program does not want to take over being the primary receiver, but simply added as an additional

receiver of the incoming messages. Use of this feature does not require that a primary receiver be defined. For every terminal, each receiver mode (except mode ZcOUTB_MLTPLX) allows a maximum of ZcMAX_SHARED_RCVRS (64) program ZLUs to be set with *action* ZcADD_SHARED.

An application program can indicate that it no longer wishes to receive messages by issuing a *zset_rcvr()* call with the same parameters as those specified in the original *zset_rcvr()* request, and setting the *action* parameter to ZcDEL_PRIMARY or ZcDEL_SHARED.

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options “-lzcom_c -lpthread” to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

action Type of action to perform.

ZcADD_PRIMARY	- Sets the program ZLU as the primary receiver.
ZcDEL_PRIMARY	- Deletes the primary receiver.
ZcADD_SHARED	- Adds the program ZLU to the list of shared receivers.
ZcDEL_SHARED	- Removes the program ZLU from the list of shared receivers.

zap ZCOM address of terminal.

mode Receiver mode:

ZcNORMAL	(0) - Set receiver for normal messages
ZcINB_MLTPLX	(1) - Set inbound multiplexer receiver
ZcOUTB_MLTPLX	(2) - Set outbound multiplexer receiver
ZcCONTROL	(3) - Set receiver for control messages

Note: Only one constant value can be specified per call for this field. You cannot logically OR these values in one call

rzap ZCOM address of the receiver program.

NOTES

Usually, an application uses only *mode* ZcNORMAL to set up itself as the receiver for normal data and status messages. An application uses *mode* ZcCONTROL only if it needs to receive "control" messages. Control messages are generated by ZCOM protocol modules. You should refer to the specific ZCOM Protocol Manual to determine the "control" messages generated by that protocol, if any.

Modes ZcINB_MLTPLX and ZcOUTB_MLTPLX require the terminal to be an inbound and outbound multiplexed terminal, respectively. Otherwise, error ZERTYPE (-54) is returned. A multiplexed terminal is usually set up by defining the proper device type in the TTGEN configuration file (using TERM or LTERM statements). It may also be modified by using *zltmx*.

When inbound multiplexing is enabled for a terminal, an inbound receiver must be set up (using *mode* ZcINB_MLTPLX) by the multiplexing program, which will be responsible for receiving messages from the terminal and passing them to the "normal receivers" kept in the logical terminal tables.

When outbound multiplexing is enabled for a terminal, an outbound receiver must be set up (using *mode* ZcOUTB_MLTPLX) by the multiplexing program, which will intercept data sent from other programs. The outbound multiplexing program is responsible for passing the correct data onto the physical terminal (using *zsend* with mode bit ZCOM_ZSEND_NOMX). For more information, see the section on Multiplexing in the *Multiprotocol ACC Programmer's Reference Guide*.

The ZcADD_SHARED action is not supported for mode ZcOUTB_MLTPLX (outbound multiplexer receiver).

RETURN VALUE

Routine *zset_rcvr* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32        ierr;
int          action;
zaddr_type   zaddr;
uint32       mode;
zaddr_type   raddr;

if (ierr = zset_rcvr (action, &zaddr, mode, &raddr)) {
          /* error return code */
}
else {
          /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zltmx(3X), zevent_rcvr(3X), zpeek(3X), zread(3X)

ZTIMR (3X)

NAME

ztimr – Enable/disable ZLU timer

SYNOPSIS

```
#include  <zcom/zcomsys.h>
#include  <zcom/zcomcall.h> /* if compiled with ANSI C (recommended) */

int32 ztimr (zap, time)

zaddr_type   *zap;
int32        time;
```

DESCRIPTION

Routine *ztimr* is used to enable or disable the timer on a program ZLU. When enabled, a timer message is added to the program ZLU queue every specified *time* seconds. Timer messages are of type **3** (see 'type' in zread).

The libraries libzcom_c.a and libpthread.a must be linked into the calling program by giving the options "-lzcom_c -lpthread" to cc(1) or ld(1).

Threads Considerations

This routine may be called from a multi-threaded application using the POSIX (1003.1c) kernel threads API package. This routine has the following characteristics when called by a multi-threaded application:

cancellation point	Thread cancellation can occur when a thread calls this routine.
async-cancel unsafe	The calling thread's cancelability type must be PTHREAD_CANCEL_DEFERRED if cancellation is enabled.
async-signal unsafe	It cannot be called from a signal handler.
fork unsafe	It cannot be called by a child process after *fork*(2) but before *exec*(2).

PARAMETERS

zap ZCOM address (program ZLU).

time Timer value in seconds (1-32767). A timer value of 0 will cancel the timer processing on the specified program ZLU. That is, no new timer messages are added to the program's input queue.

RETURN VALUE Routine *ztimr* returns **0** if successful. Otherwise, a **non-zero** error code is returned. See */opt/acc/include/zcom/zcomsys.h* for the list of ZCOM error codes and their meanings.

NOTES

A timer value of zero will cancel the timer processing on the specified ZLU. The *ztimr* call causes timer messages to be added regularly to the input queue of the specified program ZLU. A *zread* call to that ZLU will return a message type of **3** (in the *type* parameter) if it is a timer message. Thus the waiting program (or thread) is able to take alternative action, rather than waiting indefinitely for a response. If the program ZLU input queue has ten or more messages in it, the ZCOM subsystem will not add another timer message to the queue. This is to prevent a potential problem of consuming all of the ZCOM buffers with timer messages due to one or more applications not reading their input queues.

EXAMPLE

```
#include     <zcom/zcomsys.h>
#include     <zcom/zcomcall.h>

int32          ierr;
zaddr_type     zaddr;
int32          time;

if (ierr = ztimr (&zaddr, time)) {
          /* error return code */
}
else {
          /* good return code */
}
```

FILES

/opt/acc/include/zcom/zcomsys.h	ZCOM system general include file, containing data types, data structures, constants, error codes, masks, etc. Note that this must be the first include file before any other ZCOM include files.
/opt/acc/include/zcom/zcomcall.h	ZCOM routine function prototypes (requires ANSI C compilation).

SEE ALSO

zread(3X), zpeek(3X)